Computer Vision Notes

Mathematical Analysis of Computer Vision

Abhiram Kidambi

These notes are a compilation of independent reading, Wojciech Czaja's MATH416 Notes, Steve Brunton's Fourier Analysis Youtube Video Series, Shree Nayar's First Principles of Computer Vision Youtube Video Series, and College Friendly's DIP Youtube Video Series.

July 25, 2024

Contents

1 Basic Prereqs

In this section I'm going to write some basic notes on the general ideas of Computer Vision and Machine Learning that one should know before going any further in the notes.

1.1 Basic Prereqs

The basic pre-reqs that must be known to understand these notes are a fundamental understanding of linear algebra, a somewhat basic understanding calculus (and multivarius calculus), and common sense. Within linear algebra, the following topics should be known very well:

- Vectors
- Matrices
- Transformations
- Eigenvectors, Eigenvalues, and Spectral Decomposition
- Projections to Subspaces
- Column-Spaces, Row-Spaces, and Null-Spaces

A background in things such as statistics, programming, and algorithmic implementation is helpful. Many math that is "too advanced" is often *explicitly* stated as ignored or purposefully simplified for sake of keeping the notes easier to understand. At the end of each section there are resources to better understand subjects! Everything else in this section are things that can come up later down the line that will be referred to – many of these topics don't really come up until the later sections, however!

1.2 Principal Component Analysis

Principal Component Analysis (PCA) is a key tool in order to reduce the dimensions of some data based on what is presumed to be noise. We first need to define the idea of covariance as the operation which computes the product of the differences of means of two random variables at index i. We can formalize this as:

$$\text{cov}(X, Y) = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})$$

As an aside, this is also equal to the below definition using expected value, and is given as an exercise to the reader to prove except with the change of replacing n in the denominator with $n-1$:

$$\text{cov}(X, Y) = E(XY) - E(X)E(Y)$$

The change of n to $n-1$ is to make values easier to work with and "more accurate" for a lack of a better term. Do not worry about this for sake of these notes. Let's define the covariance matrix as the matrix C such that:

$$C(m)(n) = \text{cov}(X_m, X_n)$$

In other words, C is also defined as:

$$\mathbf{\Sigma} = \begin{pmatrix} \text{Cov}(X_1, X_1) & \text{Cov}(X_1, X_2) & \cdots & \text{Cov}(X_1, X_n) \\ \text{Cov}(X_2, X_1) & \text{Cov}(X_2, X_2) & \cdots & \text{Cov}(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(X_n, X_1) & \text{Cov}(X_n, X_2) & \cdots & \text{Cov}(X_n, X_n) \end{pmatrix}$$

It is not difficult to see that given a data matrix X where the columns of X are defined as input vectors, we can define the covariance matrix in the following way:

$$C_{xx} = \frac{1}{n-1} X X^T$$

Note that we are assuming that X is composed of vectors that are already 0-meaned in order to simplify the data. We can observe that C_{xx} is guaranteed to be a symmetric positive semi-definite matrix, and can therefore construct a diagonalization that consists of an orthonormal basis:

$$C_{xx} = WDW^T$$
$$WW^T = Id$$

Knowing the properties of covariance matrices, the eigenvectors capture, in order, the most amount of variance in the data. The eigenvectors show where such variance in the data is. Note to reader that this is an optimization problem provable using Lagrange Multipliers.

$$Y = WX$$

The above linear transformation based on W from the diagonalization of the covariance matrix is an orthonormal change-of-basis that maximizes the variance in the data. Since the variance is organized by eignevalue, we can perform a change of basis on X using W and note that the columns of W are the principal components. We can pick the "component" with the least change in the data, denoted as aforementioned by the lowest eigenvalue, and "remove" it from the data in order to perform dimensionality reduction.

1.3 Singular Value Decomposition

Note that although I brought up PCA using the EVD techniques (Spectral Decomposition), it is extremely common, and probably more practical, to introduce this with Singular Value Decomposition (SVD). EVD is the basic building block with which we build SVD, and to learn more about it you can take a look at the following **Youtube Series by Steve Brunton**

Before I begin, let me quickly define what **orthogonality** is in the context of matrices (this will come up in the section on the Discrete Fourier Transform as "unitary"). Any matrix which satisfies the following condition is orthogonal:

$$MM^T = I$$

This is equivalent to saying **an orthogonal transformations inverse is its transposition.** It also means that all the column vectors are perpendicular to eachother and that their magnitude is 1. Let's define the singular value decomposition of X:

$$X = U\Sigma V^T$$

We can define Σ through the singular values in the following way:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_r \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Note that the singular values are arranged in decreasing order such that $\sigma_1 \geq \sigma_2 \geq \sigma_3$ and so on so forth. We also say that the column vectors of U are known as the **left singular vectors** and the column vectors of V as the **right singular vectors**. Recall that singular value decomposition is $U\Sigma V^T$ so the column vectors of V are actually represented as row vectors in the decomposition! I won't go too far in depth in how to compute the left and right singular vectors, but for a quick overview:

- **Singular Values** - these are related to the eigenvalues of the symmetric matrices AA^T and A^TA.
- **Singular Vectors** - these are related to the eigenvectors of the symmetric matrices AA^T and A^TA

We pick which one to use based on the dimensions of the initial matrix A. We want to use the eigenvectors and eigenvalues of the matrix with higher dimensions to get all possible information.

This applies to correlation and principal components analysis because if we factorize the data matrix X into $U\Sigma V^T$, then XX^T ends up becoming $V\Sigma^2V^T$. This means that the result is an eigenvalue decomposition with a matrix in the center of the singular values squared – because of how the singular value decomposition is defined, we can simply just take the first n of the vectors in the resulting covariance matrix and leave the others because the others are less "important" or contribute less to the variation of the data. This means that using singular value decomposition gives us key insight to what **the directions or correlations** of the data are which we can use when trying to compress. **In fact, the SVD and PCA are almost the same exact thing – so much so that the PCA is called the mean-centered SVD because we subtract the mean of the data when computing the PCA.**

1.4 Convolution

The continuous convolution operator (defined as *convolving* two functions) is the following:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx$$

The discrete convolution operator is the following:

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x)g(t-x)$$

The operation of convolution is linear, and shift-invariant (which is why it is also viewed as a LSIS linear shift invariant system). This result is notable because integration and summation are also both linear and shift-invariant operations. **Convolution is also commutative and associative**. Two dimensional convolution is defined as the following operation:

$$g(x,y) = (f * h)(x,y) = \iint_{-\infty}^{\infty} f(u,v)h(x-u, x-v)\, dudv$$

$$g(x,y) = (f * h)(x,y) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(u,v)h(x-u, x-v)$$

Note that for convolution to be defined, for lack of a better terms, the functions must be "bounded". If the following:

$$\sum_{m=-\infty}^{\infty} f(m) = C$$

$$\int_{-\infty}^{\infty} f(m)dm = C$$

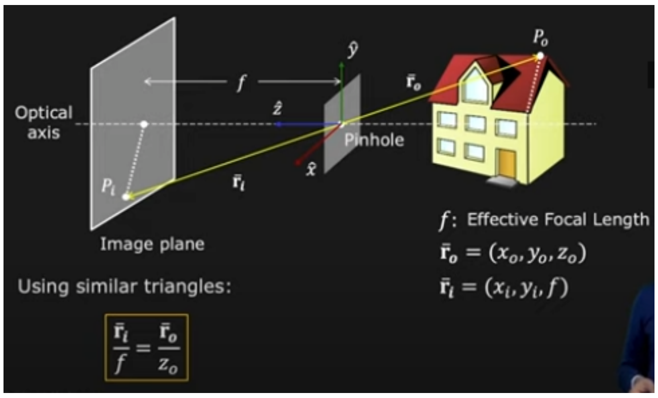
This is important because otherwise, convolution will not be defined and will tend toward infinty each time. If one of the series/integrals do not converge or decay to 0 fast enough, that produces a problem in practical considerations because convolution loses a lot of meaning. Particularly for computer vision, the discrete versions of these convolutions are significantly more important because images are often represented as matrices. A couple of important facts to know (which you might not yet understand) about convolution:

- Convolution in the spatial domain is equivalent to multiplication in the Fourier/frequency domain
- Convolution in the Fourier/frequency domain is equivalent to multiplication in the spatial domain.

These will be briefly discussed in the Image Processing section, but if you would like thorough proofs of these, please look online.

1.5 Pinhole Projection

This is important to have an understanding of what an image, at it's core, is. The main paradigm for cameras and photo-taking is the **pinhole camera model** (do NOT confuse with pigeonhole principle which is different). The pinhole model works by taking a small point (mathematically speaking, this would be infinitesimally small), and using it as an aperture to let light pass in. This projection is basically how we formulate the idea of taking a 3d image and making it appear 2d in a camera.


Optical axis
f
$\hat{y}$
$\hat{z}$
$\hat{x}$
Pinhole
P_o
$\bar{\mathbf{r}}_o$
P_i
$\bar{\mathbf{r}}_i$
Image plane
Using similar triangles:
$\boxed{\frac{\bar{\mathbf{r}}_i}{f} = \frac{\bar{\mathbf{r}}_o}{z_o}}$
f: Effective Focal Length
$\bar{\mathbf{r}}_o = (x_o, y_o, z_o)$
$\bar{\mathbf{r}}_i = (x_i, y_i, f)$

FPCV Pinhole and Perspective Projection at 2:56

Notice the logic of similar triangles which can tell that the corresponding 2d coordinate of a 3d coordinate $(X, Y, Z) \to (x, y)$ given f is the focal distance as mentioned above is:

$$x = f\frac{X}{Z}$$

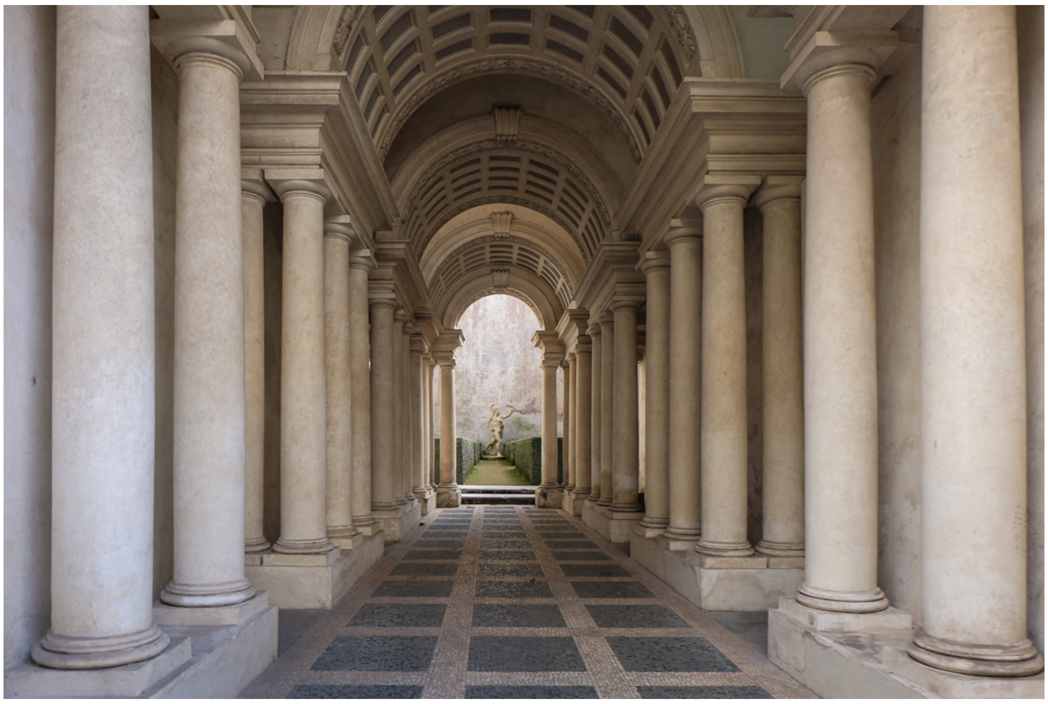
$$y = f\frac{Y}{Z}$$

We can also compute magnitude of the coordinatea from the center of the imagep lane explicitly (which I won't do here, but the main conclusion is obvious) and determine:

$$m = \frac{f}{z_0}$$

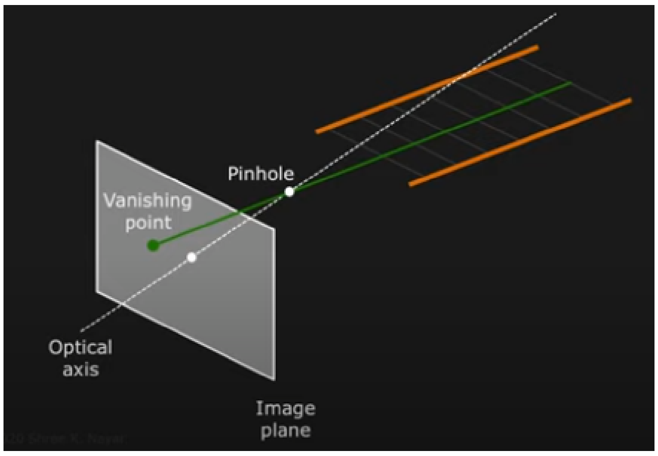
The importance of this result is that it suggests magnification is inversely proportional to the distance (which is z_0) – this means that a part of an image that is "less magnified" means that part of the image is likely farther away.

Let's also introduce the notion of the **vanishing point** which is where the image vanishes – this normally is the center of the image but it's where all the lines of depth in an image converge.



Galleria Espalda in Rome created by Francesco Borromini

The vanishing point for this image is the statue in the center – see how all the lines that appear going "into" the image stop or converge at the statue. Please note that **all parallel lines in an image have the same vanishing point**. That is to say that if the image has non-parallel lines, there can be multiple vanishing points, and that parallel lines will always have a *single* vanishing point. This vanishing point is located where the line in that parallel direction goes through the pinhole and pierces the image plane.



FPCV Pinhole and Prospective Projection at 13:38

Note that it's also very important to care for the pinhole size. Although mathematically we want the pinhole to be as tiny as possible, practically speaking this would mean no light gets to the image through the aperture. If the pinhole is too big, normally too much excess light gets in the image (creating a blurry effect) while if the pinhole is too small, **diffraction** can occur. **Diffraction** is an idea from wave physics where light, represented as a wave, bends around the aperture and less light is in the image – this is a result of the single-slit experiment. Let's discuss the **ideal pinhole diameter**:

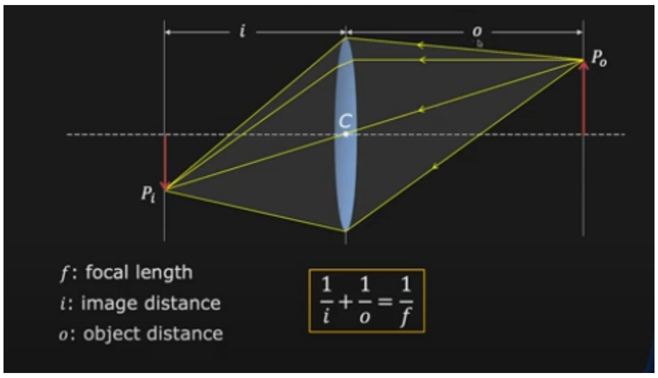
$$d \approx 2\sqrt{f\lambda}$$

$$f : \text{effective focal length}\ , \lambda : \text{wavelength}$$

We normally pick the wavelength to be the average wavelength of the image, or we can ballpark it to around $550nm$ and pick the ideal pinhole so that we make the image as clear as possible. Now, although this mathematical model for cameras works great at an abstract level, we need to introduce lenses to make it practical.

1.6 Image Formation thru Lenses

Recall that the pinhole projection works great, but it often isn't very practical because although we want all light to pass through a single "pinhole" and cross through to an image, that just means not enough light is produced. We'll get a very dark image! So, we normally use **lenses** to gather more light.



FPCV Image Formation using Lenses 1:16

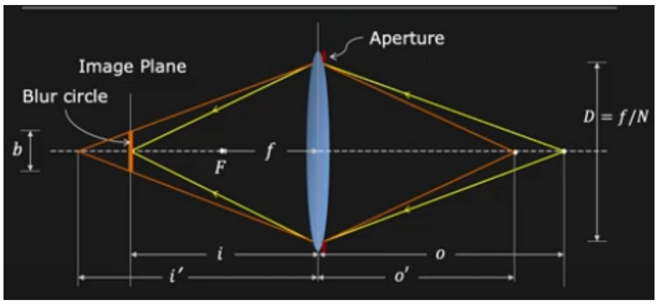
Let's take a look at the above image to understand more about lenses! In a pinhole projection model, the only ray that would have made it to the image plane would be the ray crossing through the center projection point. Let's analyze lens formulation by using the **Gaussian Lens Law**:

$$\frac{1}{i} + \frac{1}{o} = \frac{1}{f}$$

We can also use similar triangles to define the magnification (like before) as:

$$m = \frac{h_i}{h_0}$$

h_i is defined as the image height in the resulting image while h_0 is defined as the image height in the initial image. Because lenses invert the image, these two heights are normally measured in opposite directions. We can also introduce the **two lens** system which allows us to place lens sequentially in order to get a better type of image. Do note that this type of technology is used in DSLR cameras and many iPhones – if we have an even number of lenses, note that the image isn't inverted. The two lens system also enables us to have the "zooming" effect. Read more about it in the Lens Formation video by FPCV at 5:05.

h 

FPCV Image Formation using Lenses 9:09

Let's introduce the idea of **aperture** – aperture is how "open" the lens is to taking light. In the pinhole projection model, we can assume this to be, practically speaking, how large the pinhole is. Let's define aperture as d, and f as the focal length which is a constant.

$$d = \frac{f}{N}$$

N is defined as the **f-stop** (or f-number, f-ratio) of the lens. The aperture and f-stop have an inverse relationship based on the focal length (where the lens is relative to the resulting image). This means that **as the aperture/window of light gets smaller, we get less light in the image, and the f-stop increases.**

One of the prices we pay for using lenses is the idea of **lens defocus**. When we use lenses, only one plane will be *totally* in focus with the surrounding areas getting less and less in focus. Note that in the above image, only objects at a certain particular distance are actually going to be rendered as a single point in the result. Other points at different positions are going to be blurry because they won't be able to intersect at the image plane. The above image depicts the case where things are in front of the focus plane, but in the opposing case where things are behind the focus plane, the resulting rays will cross eachother and create a different blur circle. This **blur circle** is computable through similar triangles and is **proportional to diameter of aperture and inversely proportional to f-stop.** Note that we can also change the focus of an image by zooming because that changes the focal distance.

1.7 Depth of Field

This concept is very similar to what was mentioned above, but requires some more nuance in understanding the notions of digital images. We normally don't care if the blur circle is smaller than a pixel because then it is still in focus for our image. Normally speaking, we can achieve a concept called **hyperfocal distance** which is the distance at which everything beyond a certain plane in an image appears in focus because its blue circle is less than the width of a pixel. It can be calculated using the following formula:

$$h = \frac{f^2}{Nc} + f$$

We can define c as the max blur size or the width of a pixel, N as the F-Stop number (basically dependent on the aperture width), and f as the focal length. Note that f and c are fixed parameters so the F-Stop is the only changeable parameter here which means we can adjust the f-stop so any point at h distance or further is in focus. Let's quickly discuss some properties of lenses:

- **Blocking the lens** - recall that lenses don't necessarily take light solely from one particular area. It isn't the same as blocking something in real life – your lens is capable of getting information from light rays all around its physical surface area. This means that dust normally doesn't impact the lens significantly.

- **Tilting the lens** - this basically takes the plane of focus and warps it. It has a lot of uses in places where you want the plane of focus to not be parallel to the image plane but instead want it to be at an angle. It's often called a **tilt camera** and the mathematical principle behind this is called the **Scheimpflug Principle**.

2 Fourier Analysis Topics

This section is the closest to the concepts you would learn in your average DSP course you take in a math department. Material from this section is mostly taken from Czaja's MATH416 Notes, and Steve Brunton's Webseries and Textbook on Fourier-Driven Data Analysis.

2.1 Discrete Fourier Transform

WARNING: there are different versions of the DFT and CFT used in the notes, some are normalized and some aren't, most of the properties I talk about apply to BOTH transforms though and I distinguish when that isn't the case.

More important than the mathematics behind this transform, which I will get into, is understanding the **meaning** of this transform. The DFT is a transform which takes in a vector in the time-space and converts it into the frequency-space. Broadly speaking, it takes in a vector and composes it of individual discrete sinusoidal (sin and cos) functions that represent the initial signal. It can be defined the following way:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-i\frac{2\pi}{N}kn}, \quad k = 0, 1, 2, \ldots, N-1$$

Let us go through each part of the equation and define what they represent:

- X - the upper-case X represents the output of the Fourier Transform as a vector
- N - represents the dimension of the input vector
- n - represents a variable that is the index of every element of the input vector
- k - represents the index of the Fourier transform vector and varies from 0-N-1 because a Fourier transform on a vector of "N" elements will output another vector of "N" elements

Understand that we can use Euler's identity to determine: $e^{ix} = \cos(x) + i\sin(x)$ which is how the sinusoidal functions are created. Also note that the integral Fourier transform is very similar to the DIscrete Fourier Transform simply replacing the summation with an integral instead and using a different normalization procedure (fraction in front):

$$F(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)e^{-ixt}\, dx$$

Note that given the DFT, we can also use the definition of a matrix to reformulate the DFT formula given above as the following matrix:

$$F(m)(n) = e^{-imn\frac{2\pi}{N}}$$

Note that although the above definitions are valid, many (including these notes) often normalize the DFT by a division of $\sqrt{N}$ in order to preserve the unitarity of the DFT matrix so the final result of the DFT matrix is the following (unitarity is discussed later):

$$F(m)(n) = \frac{1}{\sqrt{N}}e^{-imn\frac{2\pi}{N}}$$

There are several important properties of the DFT:

- **Unitarity** - this is an extension of the idea of orthogonal matrices which satisfy the following X:

$$X\bar{X}^T = Id$$

$$\bar{X}^T X = Id$$

This is provable by taking an individual column of the DFT Matrix "n" and dotting it with itself and getting 1 and dotting it with ANY other column vector and getting 0. For a thorough proof of this, check M416 Notes on the DFT section (end of page 48 to page 49).

- **Symmetric Matrix** - note that:

$$F(m)(n) = F(n)(m) = \frac{1}{\sqrt{N}} e^{-imn\frac{2\pi}{N}}$$

Thus, we can claim that the matrix is symmetric and therefore is equivalent to its transposition.

- **Inverse is Conjugate** - note the following:

$$FF^{-1} = F * \bar{F}^T = Id$$

$$\bar{F}^T = \bar{F}$$

$$F^{-1} = \bar{F}$$

- **Parseval's Theorem and Plancherel's Theorem** - this isn't particularly important to computer vision, but this is very commonly associated with the DFT and Fourier Transforms in general so I want to mention it. Proof isn't given but this is straightforward to prove:

$$\sum_{n=-\infty}^{\infty} |x[n]|^2 = \sum_{k=0}^{N-1} |X[k]|^2$$

Note that Plancherel's theorem is similar, but rather defines a relationship for the inner product of two vectors as the inner product of the DFT of the two vectors. These properties aren't that particularly useful for CV, and don't give that much more context than the properties previously discussed, but they are so commonly associated with the DFT that any textbook/notes that introduce the DFT will mention them.

- **Periodicity: this gets a bit unnecessarily mathy feel free to skip**
An important observation in the definition of the Fourier transform for both the real and continuous case is the periodicity of the purely imaginary exponent. I will show this for the DFT, but this same approach easily translates to the CFT:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}kn}, \quad k = 0, 1, 2, \ldots, N-1$$

Note that if instead of "k", you input "N+k", you end up getting the following result:

$$X[N+k] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}(N+k)n}, \quad k = 0, 1, 2, \ldots, N-1$$

Take a look at the exponent of the e within the summation:

$$-i\frac{2\pi}{N}(N+k)n = \frac{-2\pi in(N+k)}{N} = -2\pi in - \frac{2\pi ikn}{N}$$

However note that since $n \in \mathbf{Z}$, we are guaranteed that the $2\pi in$ doesn't change the output of the exponent since:

$$e^a = e^{a+2\pi i}$$

Thus we conclude that:

$$X[k] = X[N+k] = X[k \mod N]$$

Using this information, we can determine some more "mathematical" properties about the DFT. These properties end up helping us when constructing the Fast Fourier Transform, and can allow us to express initial inputs and outputs of the Fourier transform as periodic infinite sequences which makes handling them (mathematically speaking) easier in particular situations.

A quick note on the **2 dimensional Discrete Fourier Transform**: I won't really get into proving this, but applying the DFT to a matrix simply involves applying the DFT to each of the column vectors, and then applying the DFT to each of the row vectors when the new column vectors (output of first DFT) are completed and combining it all together. This means that applying the DFT to matrices or images is actually relatively straightforward which is another reason why it is so preferred. But, there are still techniques to make it's implementation faster...

2.2 Fast Fourier Transform

Another important topic for those more interested in DSP (as opposed to necessarily computer vision) is the Fast Fourier Transform (stylized FFT). The most common approach to the FFT is the Cooley-Turkey method and it relies on the Danielson-Lanczos lemma shown without proof below (look at the MATH416 notes to see the proof):

Let N be a positive even integer and define the following sequences E and O:

$$O(n) = v(2n), n \in \mathbf{Z}$$

$$O(n) = v(2n+1), n \in \mathbf{Z}$$

Then, for all $m = 0, ..., N-1$,

$$F_N(v)(m) = F_{\frac{N}{2}}(E)(m) + e^{\frac{-2\pi i m}{N}} F_{\frac{N}{2}}(O)(m)$$

Be very careful to note that $F_N(v)(m)$ isn't the DFT but rather is defined as the m-th coefficient of the DFT of vector v:

$$F(v)(m) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} v(n) e^{-2\pi i \frac{mn}{n}}$$

In order to construct the Fast Fourier Transform, we note the important property that a DFT of a vector of length 1 is itself (because $m = 0$, $n = 0$, and $N = 1$ so $\frac{1}{\sqrt{N}} = 1$)

1. First split the input vector into two sequences O and E.
2. Compute the DFT of O and E by recursively applying the Danielson-Lanczos lemma, and use the Danielson-Lanczos lemma to combine the two to get the DFT of V.
3. Recursively compute the DFT of vector V until it becomes length 1 at which point, use the fact that the DFT of a vector of length 1 is itself.

This process is very similar (to those whom have taken an algos course prior) to MergeSort. Instead of computing an $O(n^2)$ operation through manual computation, we split it into $\log n$ computations which result in an $O(1)$ output, and run an $O(n)$ operation on the resulting computations by combining. Thus, we have been able to reduce the DFT at $O(n^2)$ to the FFT at $O(n \log n)$.

2.3 Other Important Frequency-Scale Transforms

There are many other important transforms of different types that are also used for digital signals processing (that fall in a similar umbrella as the DFT - some different transforms are discussed later!) but I don't think are that important for the average person interested in computer vision so I'll briefly mention them:

- **Discrete Sine Transform**
  This transform uses the sin function. It is a real (NON-imaginary) transform that instead of returning sinusoidal functions through complex exponentials, returns them through sin functions of varying frequencies and amplitudes. There are mostly 4 types, with type 1 being defined the following way:
  $$S(m,n) = \sqrt{\frac{2}{N}}\sin(\frac{\pi mn}{N})$$
  Note that this is not only a unitary matrix, it's entirely real and symmetric which means it satisfies $S^2 = Id$. Other variants of the DST aren't necessarily unitary/orthogonal and symmetric so be careful.

- **Discrete Cosine Transform**
  This transform uses the cos function. It is also a real (NON-imaginary) transform. There are mostly 4 types with type 1 being defined the following way:
  $$C(m)(n) = \beta(m)\beta(n)\cos(\frac{\pi mn}{N})$$
  Note that the β function is defined in order to normalize the DCT. This particular transform is also unitary, real, and symmetric.

- **Discrete Harley Transform**
  This is very similar to the Discrete Fourier Transform except avoids the need for complex numbers entirely and is defined the following way:
  $$H(m)(n) = \frac{1}{\sqrt{N}}(\cos(\frac{2\pi mn}{N}) + \sin(\frac{2\pi mn}{N})$$
  This is particularly useful because it avoids complex numbers. There are also advantageous properties that make it easier to compute things like the DFT with minimal computations at the end.

Note that there are other transforms as well but they simply aren't as common as the above. Moreover, the most important one to remember is the Discrete Fourier Transform for it is the most useful in the field of computer vision.

2.4 Uncertainty Principle

You may remember this from Quantum Mechanics as the Heisenberg Uncertainty Principle! I'm not totally sure if it's equivalent, but from my understanding, it's essentially the same *concept.* The idea briefly stated is that: you can never precisely know both the exact time-components and frequency-components of a signal. This means that there will always be a trade-off between which one you want to use. The equation that governs this is:

$$\Delta t * \Delta w \geq \frac{1}{2}$$

Note that Δt is the uncertainty of time and Δw is the uncertainty of frequency. In other words, this means that given a pixel in an image, we can NOT (and will NEVER be able to) identify the exact frequencies at a particular pixel. We have to either trade-off knowing the exact frequencies for "around the exact" frequencies or the pixel for "a general region of the pixel".

2.5 Gabor Transform and Spectogram

This is a technique we can use in Fourier analysis to get around the uncertainty principle issue. To understand and make sense of it in this context, however, it makes a lot of sense to think about audio signals. The broad gist of the Gabor transform is to convolve a normalized Gaussian function (area under bell curve is 1) with the Fourier Transform of a signal. The idea is that since we can't necessarily know the exact frequency at a particular "pixel" or particular "time", why not take compute it in the general (normalized Gaussian region with that point at the center) vicinity of the area of a pixel, and get some broad information about the frequencies in that area near the pixel.

Define a function $g(t)$ which is a normalized Gaussian function. It is in general, the window function and you can pick any other normalized function as well to be used here; we just need it to be centered at t. Define the Gabor transform as the convolution of the function g and the Fourier Transform to obtain the following for the continuous case:

$$G(f) = \hat{f}_g(t, w) = \int_{-\infty}^{\infty} f(x) e^{-wt} g(x - t) \text{``}dx$$

Please note that the Gabor transform is a function on the signal f, but produces a function which takes in an input of both time and frequency. It is essentially asking the question, given a time "t", what is the strength of a given frequency at "w", and is plotted either in some type of color, or shaded by darkness. The resulting output when plotted in a graph is known as a spectrogram or sonograph. This is the same algorithm things like Shazam and Apple Music use to find and determine music.

This topic will end up coming back again as the Gabor Filter which is a discrete version of this transform that is used for texture-analysis in images.

2.6 Filtering Transforms

These transforms are VERY important for computer vision since they allow you to filter through data and determine what type of frquencies, in particular you need. To start, we note that there are many different types of filter transforms:

- **Low-Pass Filter** - retains low frequencies while high frequencies are suppressed
- **High-Pass Filter** - retains high frequencies while low frequences are suppressed
- **Band-Pass Filter** - retains frequencies in a specified frequency band
- **Band-Stop Filter** - frequencies in a specified frequency band are suppressed
- **Notch Filter** - suppresses a single frequency

We say that a sequence $f = \{f(k)\}_{k \in \mathbf{Z}}$ is defined if the smallest interval which contains all integers $k \in \mathbf{Z}$ such that $f(k) \neq 0$ can be marked as $[a, b]$ where $a, b \in \mathbf{Z}$. In practice, it's basically saying that the filter needs to NOT be infinite. We can first define the sequence u (the vector that is to be filtered) as a function $u = \{u(k)\}_{k \in \mathbf{Z}}$. The associated filter transform is:

$$\forall n \in \mathbf{Z},\ F(u)(n) = \sum_{k \in \mathbf{Z}} f(k - 2n) u(k)$$

Notice that this is *very* similar to the idea of discrete convolution which is defined the following way:

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x) g(t - x)$$

Note that if we define $F(u)$ as it's own function, which it is in some sense (because it is just a function that outputs the result at the n-th position of the vector given the filter transform and sequence u), we can note that the major differences are that the filter transform uses $2n$ instead of n (where the convolution uses t) and the filter transform doesn't subtract k from $2n$ when convolution would subtract x from t.

This basically suggests that the filter transform doesn't "invert" or "flip" the input (like a convolution would) and "filters" through the vector by a series of 2. This is because of how the Wavelet transform is defined and because in DSP, there is an advantage to using the filter transform to reduce the dimension of the dataset down by a factor of 2. Here is an example to better understand the Filter transform:

EXAMPLE: *Suppose we consider a filter f on length 4 supported on $[0, 3]$ then for input signial u which also has length 4 supported on $[0, 3]$, show what the matrix equivalent of the filtering transform is.*

$$\forall n \in \mathbf{Z},\ F(u)(n) = \sum_{k \in \mathbf{Z}} f(k - 2n)u(k)$$

$$F(u)(1) = f(0-2)u(0) + f(1-2)u(1) + f(2-2)u(2) + f(3-2)u(3) = f(0)u(2) + f(1)u(3)$$

$$F(u)(0) = f(0)u(0) + f(1)u(1) + f(2)u(2) + f(3)u(3)$$

$$F(u)(-1) = f(0+2)u(0) + f(1+2)u(2) + f(2+2)u(2) + f(3+2)u(3) = f(2)u(0) + f(3)u(1)$$

Given that we are trying to create a matrix formulation that fits the equation: $F\vec{u} = G$ where G is the output of the Filter Transform:

$$\boxed{F = \begin{pmatrix} f(2) & f(3) & 0 & 0 \\ f(0) & f(1) & f(2) & f(3) \\ 0 & 0 & f(0) & f(1) \end{pmatrix}}$$

Additionally, note that we can use a discrete substitution and derive the following equivalent formula for the filter transform:

$$F(u)(n) = \sum_{k \in \mathbf{Z}} f(k - 2n)u(k) = \sum_{k \in \mathbf{Z}} f(k)u(k + 2n)$$

Some interesting and semi-important facts about the Filter Transform.

- Given a filter f supported on the interval $[a, b]$ and the sequence u is supported on the interval $[x, y]$, then we know that $F(u)$ is supported on the following interval (MATH416 notes for proof):

$$\left[\lceil \frac{x-b}{2} \rceil, \lfloor \frac{y-a}{2} \rfloor\right]$$

- The Filter Transform often leaves behind a lot of zeroes (especially in the starting and ending stages when the filter "starts" to interact with the vector. For sake of computer science and making more efficient transformations, mathematicians have developed a way to use the periodicity of a filter's application to a signal to reduce this. You can read more about this in the MATH416 notes in pages 87-88. If you're into understanding patterns, the general gist is that for the above example, instead of the matrix F, it would output G:

$$G = \begin{pmatrix} f(0) & f(1) & f(2) & f(3) \\ f(2) & f(3) & f(0) & f(1) \end{pmatrix}$$

2.7 Orthogonal Conjugate Quadrature Filters (OCQF)

We need to assume the following properties of a given filter to assign it to the term "low pass filter":

- **Finiteness** - Assume that ψ has a compact support, then the sequence $\{h(k) : k \in \mathbf{Z}\}$ is different from 0 for all but finitely many k's (recall this is a condition I mentioned earlier in the definition of a filter since it doesn't really make sense to process finite data (image-related data especially) with an infinite filter.
- **Self-Orthonormality** - let us first define σ as the Kroenecker Delta Function:
$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$
Then the condition is the following:
$$\forall m, n \in \mathbf{Z},\ \sum_k h(k + 2n)h(k \bar{+} 2m) = \sigma_{mn}$$
This is basically suggesting that the filter's impulse (or output) is orthogonal to any of its time-shifted variants and since we defined the filter transform as "separated by units of 2" (recall the "+2n" in the definition that I mentioned earlier).
- **Normalization** - assume that ψ is integrable then we can say that the sequence $\{h(k) : k \in \mathbf{Z}\}$ satisfies the following conditions (second is easily derived from the first):
$$\sum_k h(2k) = \sum_k h(2k+1) = \frac{1}{\sqrt{2}}$$
$$\sum_k h(k) = \sqrt{2}$$

Upon defining such a low-pass filter, which is *ANY* filter which fits the above properties, we can then claim that there exists a unique associated high pass filter defined by the following where m is the length of the filter:
$$g(k) = (-1)^k h(m \bar{-} k)$$
The associated high-pass is guaranteed to exhibit the *normalization* and the *self-orthonormal* conditions mentioned earlier. We are also guaranteed independence (the two filters are orthonormal with respect to eachother) and completeness:

- **Independence**
$$\forall m, n \in \mathbf{Z},\ \sum_k g(k + 2n)h(k \bar{+} 2m) = 0$$
- **Completeness**
$$\forall m, n \in \mathbf{Z},\ \sum_k h(2k + n)h(2\bar{k} + m) + \sum_k g(2k + n)g(2\bar{k} + n) = \sigma_{mn}$$

All these properties are relatively straightforward to prove, and can be shown in the MATH416 notes in the section on Wavelets and Multiresolution Analysis.

Let us now define **Orthogonal Conjugate Quadrature Filters** as a pair of Low-Pass Filters and High-Pass Filters that meet the above criterion. Note that if we can find *any* filter that is a low-pass filter, we can use the formula defined earlier to create the associated high-pass filter which will satisfy the Independence and Completeness properties. It is also easily provable that an OCQF must have an even length.

2.8 Wavelets and Multiresolution Analysis

This is a very niche topic that is useful for computer vision, but can be very bogged down with math so I want to be careful. It is also incredibly important for image-compression and is used in the new JPEG compression algorithm.

Let us define the idea of wavelets by first acknowledging the Uncertainty Principle - we already accept we can never know *exactly* when a frequency takes place. But an important secondary conclusion about human and image nature: **lower frequencies tend to last for longer time while higher frequencies tend to last for shorter time**. It is only natural to define a system of compression/analysis that uses this information. Let us start by defining properties of a **wavelet function**:

1. **Zero-Mean**:
$$\int_{-\infty}^{\infty} \psi(t)\, dt = 0$$
2. **Normalization**:
$$\int_{-\infty}^{\infty} |\psi(t)|^2\, dt = 1$$
3. **Decay/Finite-Support**: The wavelet must be non-zero for only a finite amount of time within the wavelet or else it won't be able to capture features based on isolated frequency inputs. There are also connections to an idea called the Fast Wavelet Transform which requires the wavelet decays or has finite support.
4. **Oscillatory**: Although not necessarily a requirement, practically speaking, all wavelet functions must be oscillatory in order to capture the high frequency components of a signal.

We can also define the notion of child wavelets from the mother wavelet below. These essentially take the wavelet function and shrink it's behavior to half of where the original wavelet function was defined, leaving the other half 0. It also adjusts properties of the functions (time-shifting and amplitude-shifting) in order for it to meet the above properties.

$$\psi_{ab}(t) = \frac{1}{\sqrt{a}}\psi(\frac{t-b}{a})$$

Some examples of wavelets (along with a brief description are the following):

1. Haar Wavelet - a function that is 1 for half the input region, and -1 for the other half, and 0 everywhere else. This wavelet is most commonly associated with the Discrete Haar Transform commonly used in image compression.
2. Daubechies Wavelet - a weird-looking shark-tooth wavelet function also used for image compression, but also used for image-analysis.
3. Mexican Hat Wavelet - looks like a sombrero lol, it is used along with many other filters for other particular reasons, but I don't really care enough to write or find them out

We define the Discrete Wavelet Transform through using the idea of a "pass". Broadly speaking, we want to use a filter transform to filter the data into what is "high frequency" and "low frequency" using the Orthogonal Conjugate Quadrature Filter scheme. Let us define the filter transform for the low pass filter as G and the high pass filter as H (matrices are capitalized, vectors are lower case):

$$\begin{bmatrix} G \\ H \end{bmatrix} \vec{v} = \begin{bmatrix} \vec{a} \\ \vec{d} \end{bmatrix}$$

This means that **a single pass** of the Discrete Wavelet Transform applied to a vector $\vec{v}$ of length 2^m would yield a vector of length 2^m composed of two vectors $\vec{a}$ and $\vec{d}$ each having 2^{m-1} elements. It is worth noting that:

$$G\vec{v} = \vec{a}$$
$$H\vec{v} = \vec{d}$$

We say that vector $\vec{a}$ composed of the result of the filter transform with a low-pass filter defined as matrix G creates the **approximation coefficients** while the vector $\vec{d}$ composed of the result of the filter transform with the high-pass filter defined as matrix H creates the **detail coefficients**.

The key observation here is that the low-frequency coefficients are the important ones that last for longer and contribute more to the visuals of the image while the higher-frequency coefficients are either noise, or make the image look slightly better, and so can be mostly ignored. If you're asking about why the Uncertainty Principle matters here, it sort of doesn't... it's more important for running the wavelet transform on audio signals in the continuous case.

2.9 Full Discrete Wavelet Transform

The Full Discrete Wavelet transform *(for a vector, NOT an image)* revolves around the algorithm above, but simply recursively applied until we end up with a single value for the Approximation Coefficient. Suppose we run a single pass of the Discrete Wavelet Transform on vector $\vec{v}$ of length 2^m:

$$\begin{bmatrix} G_m \\ H_m \end{bmatrix} \vec{v_m} = \begin{bmatrix} \vec{a_{m-1}} \\ \vec{d_{m-1}} \end{bmatrix}$$

We now redefine *NEW* matrices G and H that are intended for filtering the output of the *approximation* coefficients, and then complete a *NEW* pass of the Wavelet Transform on the output of $\vec{a}$. So, for the second pass, we use the vector $\vec{a}$ from earlier and continue:

$$\begin{bmatrix} G_{m-1} \\ H_{m-1} \end{bmatrix} \vec{a_{m-1}} = \begin{bmatrix} \vec{a_{m-2}} \\ \vec{d_{m-2}} \end{bmatrix}$$

The above is the 2nd pass of the wavelet transform, and we keep on doing this same "pass of the wavelet transform", yet now to a_{m-2} adjusting the matrices accordingly. We keep on doing this m times until we obtain a single value for the coefficient and say that the output of the Full Discrete Wavelet Transform is the vector:

$$\begin{bmatrix} a_0 & d_0 & d_1 & \dots & d_{m-2} & d_{m-1} \end{bmatrix}^T$$

This produces a vector of length m composed of *vectors* (nested vectors!) that are the output of the Full Discrete Wavelet Transform. Broadly speaking, the first approximation coefficient defines the broad "approximation" or "basis" of the signal. The remainder of the detail coefficients operate at different *resolutions* since they are at different vector-lengths and were separated (high from low) at different "passes" or "iterations" of the transform.

Warning: these matrices G and H aren't actually using the same filter transform I mentioned earlier but rather the periodic filter transform to keep dimensions in check and simplify computation – check that out in the MATH416 notes if that interests you.

2.10 Full Discrete Haar Transform

The Discrete Haar Transform, is just a more specific Discrete Wavelet Transform defined with the particular Haar Filter. The Haar Filter is the set of following filters (low-pass first, high-pass second):

$$h = \left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$$

$$g = \left[\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]$$

If we use the first filter as the low-pass filter and the second filter as the high-pass filter, and use the OCQF together in order to perform the Discrete Wavelet Transform, we have computed the Discrete Haar Transform. Look at the MATH416 notes for an example here. That being said, this whole discussion on the transforms should merit a really important question: **what do the wavelet functions I mentioned about in section 2.5 have anything to do with this transform?** This is something I need to answer later, but broadly speaking, it's related to how "projecting" the vector on the functions works.

2.11 2 Dimensional Wavelet Transform Example

Here is where things become a lot more fun! Let us first discuss how the 2 dimensional wavelet transform works from an idea-perspective, and then see it by taking a look at **a Jupyter Notebook I made.** Instead of constructing a "low-pass filter" and "high-pass filter" which separate a *vector* into low-frequency and high-frequency components, we instead run them in horizontal and vertical directions:

- **LL Coefficients** - these are the coefficients that arise when you run low-pass filters in both directions (on the column vectors AND the row vectors). These are the same thing as approximation coefficients, and tend to carry the most amount of "information" about the image.
- **LH and HL Coefficients** - these are both the exact same things but in different directions. LH is when you apply the "low-pass filter" across the rows but a "high-pass filter" across the columns. This will pick out the features that stay the same across the rows, but not the columns. HL is the exact opposite, and will pick out the exact opposite features. These are both considered part of the detail coefficients.
- **HH Coefficients** - these are coefficients that are produced when you run high-pass filters for both the column and row vectors. This tends to have the least amount of information compared to the other coefficients. It is also considered a part of the detail coefficients.

2.12 Laplace and Z Transform

This is an aside because these transforms are often given similar functional-names to the Fourier transform ("transforms from the time domain to the frequency domain"), but aren't necessarily exactly the same thing.

$$\mathcal{L}\{f(t)\} = F(s) = \int_0^\infty f(t)e^{-st}\,dt, s \in \mathbf{C}$$

$$\mathcal{Z}\{x[n]\} = X(z) = \sum_{n=-\infty}^{\infty} x[n]z^{-n}, n \in \mathbf{C}$$

Let us first compare the Laplace and the Continuous Fourier transform, here they are repeated but next to eachother:

$$\mathcal{F}\{f(t)\} = F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt\,,\ \omega \in \mathbf{R}$$

$$\mathcal{L}\{f(t)\} = F(s) = \int_{0}^{\infty} f(t)e^{-st}\,dt\,,\ s \in \mathbf{C}$$

Here are some key differences to note between the two transforms:

- The integrals are different because the Fourier Transform's integral varies from $-\infty$ to ∞ while the Laplace Transform's Integral varies from 0 to ∞.
- The Laplace transform takes in any complex number $s \in \mathbf{C}$ while the Fourier transform takes in any real number $\omega \in \mathbf{R}$ and multiplies it by i to use in the exponential.

Overall, you should be feeling that the transforms are *very* similar! Let us show how the Laplace transform is in some sense an extension of the Fourier Transform, and let's start by recognizing a central limitation of the practicality of the Fourier transform: **in order to apply the Fourier Transform to a function, we require that the function it is being applied to decays on both tail ends.** This is because otherwise, the integral we compute simply would be unbounded. This condition is often called the *finite energy* condition and is stated the following way where $f(t)$ is the function:

$$\int_{-\infty}^{\infty} |f(t)|^2\,dt < \infty$$

What happens if we want to compute the Fourier transform of a function that *doesn't* fit this criteria? Let's start by forcing the function $g(t)$ to fit the criteria of *finite energy*. Let's first define the Heaviside function:

$$h(t) = \begin{cases} 0, & \text{if } t < 0 \\ 1, & \text{if } t \geq 0 \end{cases}$$

Next, let's note that any function which is unbounded on positive infinity can be multiplied by $e^{-\lambda t}$ in order to "bound" it to 0. We can adjust λ as needed to "force" it to 0 for most functions. Let's start by defining $f'(t)$ which uses the above fact:

$$f'(t) = e^{-\lambda t} f(t)$$

But what if it accidentally blows up the negative portion of the function $f(t)$? If that doesn't already decay, or if it decays at a slow enough rate, $e^{-\lambda t}$ will augment the function to infinity. And so, we use the Heaviside function here to basically ignore all of the negative region of the function $f(t)$ regardless of whether it decays or blows to infinity for simplification purposes! Let us now define the function $g(t)$ which is the new "function" which fits the finite energy criteria, and then apply the Fourier Transform (defined as $\mathcal{F}$) on that function:

$$g(t) = e^{-\lambda t} f(t) h(t)$$

$$G(t) = \mathcal{F}(g(t)) = \int_{-\infty}^{\infty} g(t)e^{-i\omega t}\,dt\,,\ \omega \in \mathbf{R}$$

$$\int_{-\infty}^{\infty} e^{-\lambda t} f(t)h(t)e^{-i\omega t} = \int_{-\infty}^{\infty} f(t)h(t)e^{-\lambda - i\omega t}$$

Note that integrating over the Heaviside function $h(t)$ is essentially the same as integrating over the region of 0 to ∞ because the negative region is multiplied by 0:

$$\int_{0}^{\infty} f(t)e^{-\lambda - i\omega t}$$

Now we can note that the λ was defined from the $e^{-\lambda t}$ used to "forcefully" decay functions which didn't already decay to meet the finite energy requirement. But also note that the λ can be viewed as an input parameter. Let us arbitrarily define $s \in \mathbf{C}$ such that:

$$s = \lambda + i\omega t$$

We can then redefine the Laplace transform per its initial definition that is:

$$\mathcal{L}\{f(t)\} = F(s) = \int_0^\infty f(t)e^{-st}\,dt, s \in \mathbf{C}$$

We can also note that the Laplace transform is simply the Fourier transform for positive-only signals except where the input s is restricted to the unit circle and given as an input w where $e^{iw} = s$.

You might be asking, what about the Z-transform though? The Z-transform is, broadly speaking, the discrete equivalent to the Laplace transform, and we can suggest a similar relationship between the Laplace and the Fourier transforms, between the Z and Discrete Fourier transforms. However, note that in discrete signals, the *finite energy* condition is already dealt with (because we only care about N of the coefficients). That being said, the Z-transform allows us to use a more broad definition of the phrase "transforms from the time domain to the frequency domain" because the Z-transform, like the Laplace transform, allows for complex frequencies.

The importance of this is mostly in electrical engineering and differential equations where Laplace and Z transforms are used to simplify partial and ordinary differential equations, and also simplify and compress certain types of signals. For the purposes of someone interested in computer vision however, this topic is largely speaking, *practically useless.*

2.13 Practice Problems

These are some compiled practice problems focussing on ensuring understanding of theory and implementation:

1. Prove/compute the following properties (questions are arranged easiest to hardest):
   - Prove that the pair of Haar filters is the only OCQF of length 2.
   - Compute the eigenvalues of the DFT matrix (*HINT: use the unitarity of the DFT*)
   - *G*iven $v \in \mathbf{R}^N$ and $v(n) = v(-n)$ when periodically extended to $n \in \mathbf{Z}$, show the DFT of such a vector v is real-valued and even.
   - Prove the Danielson-Lanczos Lemma for the Fourier Transform.
2. Compute the following using MATLAB or Python (ideally in a Jupyter Notebook) – most of these implementations can be found **here**. Note that some of them are in MATLAB and others are in Python for convenience.
   - The Discrete Fourier Transform of an Image (manually implement it)
   - Use either Python or MATLAB's implementation of the Fast Fourier Transform and compare the speed of your DFT algo and the FFT
   - The Discrete Wavelet Transform of an Image (don't write the algorithm yourself)

2.14 Further Learning

There are so many good resources to learn more about Fourier-Based Analysis (especially given that my notes is more intended for people with some background in the field). Here are a list of some of the resources I recommend to use:

- **MATH416 Notes by Dr. Czaja** - I personally found these helpful overall, but a lot of the section on DWT, MRA, and Filters, was bogged down with a bunch of math to a point where *none* of the intuition made sense to me. It's overall a good resource.
- **Steve Brunton's Webseries** - a link to the webseries is **here**. This webseries is overall pretty good. I didn't really watch the first few videos because I knew the topics, but the focus of the series is WAY less on computation and significantly more on understanding the concepts. It's particularly noticeable in the sections on MRA, Wavelets, and Laplace.
- **Nathan Kutz Video on Wavelets** - this single video was *very* helpful for me and can be found **here**. This video talks about the intuition for the Wavelet Transform and is a really good segue from the Gabor transform to the Wavelet transform.
- **Kutz and Brunton's Textbook** - the above two mathematicians/engineers are faculty at the University of Washington and co-wrote a textbook (of which the second chapter mostly) was used to create the webseries on Fourier Analysis that I mentioned earlier. You can find a link to it **here**. I didn't use it particularly, but I'm sure it's very helpful.
- **Rafat's Website** - this website is particularly helpful if you want a more theoretical treatment, and a bit more logic on some other stuff. I didn't really use it that much, but once again, it definitely could be of help! Find a link **here**.

3 Image Processing

This is arguably, one of the most important parts of computer vision. Many of the topics in this section do rely on some knowledge of Fourier-based Analysis, but I will refer to the specific sections and cases where this happens.

Many of the notes on this topic are particularly from Shree K Nayar's First Principles in Computer Vision Image Processing series and also College Friendly's introduction to Digital Image Processing video series. I also use some textbooks like Szeliski's textbook. You can find all of these in the final section.

3.1 Introduction to Image Processing

Image processing is, broadly, the practice of preprocessing and applying various filters and transforms to an image to extract meaningful features. This involves running different algorithms, filters, kernels, and transforms on the image to identify and highlight certain characteristics. In a machine-learning pipeline, this would typically be the *very* first stage. It often encompasses both removing noise from the image, and extracting the important features. You may, however, see the term "digital image" thrown around here and there. This will be explained a bit later, but think of this, as your standard pixelated image.

We also need to formulate a way to understand *how* to process an image, and so we introduce a **simple image formulation model**:

- **Light Source** - in order to take an image, we need to have a light source prevalent. We claim that an image can be thought of as the matrix F with parameters x and y representing a specific location in the pixel. The function f represents the intensity at a particular pixel but as a function with two inputs instead. Thus:

$$f(x, y) = F(x)(y)$$

$$0 \leq F(x)(y) < \infty$$

$$0 \leq f(x, y) < \infty$$

  We say the matrix/function F/f is characterized by the product of the **source illumination** $i(x, y)$ and the **illumination reflected** $r(x, y)$. I will only show this with f even though it is the same for F. The below are definitions of the functions (note that reflection is simply a number between 0 and 1):

$$0 \leq i(x, y) < \infty$$

$$0 \leq r(x, y) \leq 1$$

$$f(x, y) = i(x, y) * r(x, y)$$

- The rest of the steps will be spoken about later but, depending on what definition of "simple image formulation model" we use, can include a *LOT* of different other steps as well.

3.2 Overview of Image Processing

Here are the different stages and terms regarding steps in image processing which we should be aware of:

- **Image Acquisition** - this is the process of taking a camera, clicking a photo (so the science behind that), and transforming the continuous image into a discrete one which we can use for image processing.
- **Image Enhancement** - this is the process of manipulating an image so the result is more suitable to find features in our models. Often times, this is chosen by the human who is running the processing; the choice of what "enhancements" to use is often subjective.
- **Image Restoration** - this is the process of improving the appearance of an image. Note that this is very similar to enhancement, except often requires more probabilistic and mathematical analysis as opposed to subjective human choices. The focus on this is also more relevant to cases where things have been distorted; assume a picture that is very blurry, we might use image restoration in order to make the image sharper again.
- **Morphological Processing** - this is the process of applying some type of transformation/adjustment to structures within an image. A great way to imagine this is thinking about a black-and-white image, and trying to either scale, dilate, erode, or perform any other transform on that structure, in the image.
- **Image Segmentation** - this is one of the *hardest* things to do in Image Processing but is the idea of partitioning an image into constitutent parts or objects. Think about taking a photo of an apple in the background, the idea of partitioning the apple from the rest of the image is a part of the image segmentation task.
- **Object Recognition** - this is again another really *hard* tasks to do in image processing and involves labeling the constitutent objects with some type of predecided label. If we were using the previous example, separating the apple from the rest of the photo falls under the umbrella of image segmentation, but labeling that separate "object" as an apple falls under the realm of object recognition.
- **Feature Selection** - this is the idea of picking a representation of an image in order to discern and pick particular features (ideally for some type of model). Choosing to use a boundary-based representation, or choosing to use a grayscale-based representation, or anything in this general field, is something that is very important yet very challenging.
- **Compression** - I won't spend much time on this particular part of processing because I discussed many of the techniques in the **Fourier Analysis Topics** section.
- **Color Image Processing** - this involves using color and analyzing the color of certain things in an image to extract features, and perform other image processing functions mentioned earlier.

3.3 Sampling and Quantization

Cameras often are a product of different light sensors, and the images they first create are called **continuous images**. These are images that aren't discretized, and can't necessarily be "turned" into pixels. With that, we want to define the notion of sampling from a continuous image and discretizing it. Let's first start by trying to discretize some type of wave function. Suppose a simple sin function of the following form:

$$f(x) = \sin(2x)$$

For the above sinusoidal function, we note that the amplitude is 1 and the period is π. If we were to sample this function along the x domain at points $n\pi$ where $n \in \mathbf{Z}$, we would get 0 every single time:

$$f(n\pi) = \sin(2(n\pi) = \sin(2n\pi) = \sin(0) = 0$$

Now, suppose the following function:

$$g(x) = \sin(\frac{x}{2})$$

If we were to sample this function at a rate of $x = n\pi$ like above, we would instead get:

$$g(n\pi) = \sin(\frac{n\pi}{2})$$

Note that for any given $n \in \mathbf{Z}$, this value is distinct only from $[0, 4]$ and periodic modulo 4 afterward, and so, we ultimately conclude that 4 different values can be sampled which can be written as the following vector:

$$[\sin(0), \sin(\frac{\pi}{2}), \sin(\pi), \sin(\frac{3\pi}{2})]^T$$

This is particularly important because it shows how sampling a continuous sinusoidal/oscillating function can lead to aliasing the data – sometimes we get pretty nice representations of what the data looks like, but other times, we end up getting some crazy information like getting repeated 0's. In more extreme cases, you can end up getting more unwanted frequencies and other information which can really screw with "discretizing" the continuous signal.

This whole field of trying to understand sampling from a continuous signal falls in the realm of electrical engineering, signals processing, and information theory, and the famous **Shannon-Nyquist Sampling Theorem** dictates how should approach sampling a continuous signal in order to avoid aliasing:

$$f_s \geq 2f_{max}$$

In this above formulation, f_s denotes the sampling ***frequency*** (NOT the period of sampling which is what I used above), and f_{max} denotes the highest frequency prevalent in the signal. Using this inequality for figuring out optimal frequencies to sample at allows for *perfect reconstruction* – we are capable of getting rid of aliasing because we are setting a lower bound for sampling so we can't "lose" components of the signal like we did for the $\sin(2x)$ case where naively looking at the sampling data would make us think we had an amplitude of 0 because all the data was 0.

We can now define the notion of a Shah function, which is a train of impulses of length 1. Think of it is as a discrete function which for periodic X, returns 1, and for all other X, returns 0:

$$s(x) = \sum_{n=-\infty}^{\infty} \delta(x - nx_0)$$

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

Note that this is a different Kronecker Delta function than the one used earlier with the subscripts. We can also define the Fourier Transform of the Shah Function as the following function (computation isn't shown, but it isn't hard to prove):

$$S(u) = \frac{1}{x_0} \sum_{n=-\infty}^{\infty} \delta(x - \frac{n}{x_0})$$

Note that the Fourier Transform of the Shah Function is *basically* a Shah Function itself except with the amplitudes adjusted (use a discrete substitution for $x' = \frac{1}{x_0}$ and adjust the Kronecker

Delta function to use $\frac{1}{x_0}$ instead of 1). That being said, we can reintroduce sampling as the Shah function in terms of multiplication. Suppose we want to sample at some x_0 rate a function $f(x)$:

$$g(x) = f(x)s(x)$$

The resulting function $g(x)$ returns a discrete signal sampled at the particular signals. We can take the Fourier transform of the whole thing and note that multiplication becomes convolution in the Fourier Domain:

$$F_s(u) = F(u) * S(u)$$

Note that since the $S(u)$ produces a delta-like function, we can note that we're simply going to get multiple copies of the sampled function over and over again! I won't get further into here, but this notion is how we define the Shannon-Nyquist Sampling Theorem, and how we work on reducing aliasing from occurring in any image. The techniques to reduce aliasing (which stem from this entire concept) are as follows:

- **Band Limit** - we run a low-pass filter on the frequencies entering the camera, and get rid of any of the frequencies which exceed half the sampling frequency (that way the maximum simply will always meet the Shannon Sampling Theorem)
- **Box Filter** - we can also run a general box-filter on the image first (generally a smaller one) to get rid of the really high frequencies prevalent in the image and reducing them down to those allowed by the filter.

Let us now define the notion of **quantizing**, which is pretty simple: it consists of just taking a continuous image, and turning it into "pixels". This is done by taking numerous horizontal lines through the image, sampling the lines and image through the techniques mentioned above, and taking the discrete sampled values and assigning them to pixels. This often constructs an "accurate enough" representation of the image which we humans can't differentiate from the initial image.

3.4 Adjacency and Distance Measures

This topic is mostly logical – in a standard Digital Image Processing exam, you'd find content on this, but I don't think it's *that* important. Let's start by defining distance measures:

- **Euclidean Distance** - the below formula gives the distance of a point (x,y) from the origin:

  $$D(x, y) = \sqrt{x^2 + y^2}$$

  For those interested in mathematics, this is also known as norm-2 – read more about norm functions from a linear algebra perspective **here**
- **Manhattan Distance** - the below formula gives the distance of a point (x,y) from the origin:

  $$D(x, y) = x + y$$

  This is norm-1 for those interested in a linear algebraic approach to norms.
- **Chessboard Distance** - the chessboard distance is basically asking, given a king places on a given pixel (x,y), how many steps would it take to get it back to 0 – this isn't necessarily computable via a formula because it depends on the structure and the distance of where the pixel is located.

Now let's discuss adjacency measures. There are typically 3 major adjacency measures used in image processing, and I'll discuss some nuances with respect to them later:

- **4-adjacency** - these are the pixels that are directly up, down, left, and right (by one pixel only) to a given pixel
- **8-adjacency** - same logic as above, except pixels that are just one "chess-step" (by a king) away from the given pixel so diagonals are included too
- **Mixed-adjacency** - these are the set of pixels that are all connected as a region given that there is at least one 4-adjacency between them (so a set of three diagonals on its own wouldn't meet the criteria for mixed-adjacency). It kind of combines 4 and 8 adjacency to look at broader amounts of pixels, but *importantly*, makes sure no ambiguities are included so there must be at least one four-adjacency connection.

These pixels are often considered "connected" if they match one of the other pixels (so in a binary image, if only two out of the 4 pixels from the 4-adjacency are the same "type" or "color" as the initial pixel, *only* those are considered 4-adjacent). When we have more "real" grayscale images, we define this notion by picking "differences" we will allow. So, we can construct a set V:

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

We say that a pixel is 4-adjacent (using the V above) if it's difference from the initial pixel (in intensity) is in the set of V and it's located either directly up, down, to the left, or to the right of the pixel.

3.5 Operations on Images

Note that intensity values are defined in some range, and the values *must* stay in that range. And so, we define the following important notions of operations particularly for grayscale images (these properties are easily translatable to properties for RGB images as well):

- **Clipping** - If we ever end up doing some type of operation on a pixel and get a value that is greater than 255, we clip the value at 255. The same thing happens if a value is less than 0. The effect this produces is often termed **saturation** (it is also used as a synonym for clipping).
- **Round Floating Numbers** - always round floating numbers of intensity values to the nearest integers.
- For anything else that happens in crazy instances, just use best judgement. If you end up getting a value like infinity, it often makes sense to either arbitrarily assign that to 0 or 255. Using common sense here is pretty much the common approach.

We can then define the notion of point-wise operations. Suppose we have two images F and G with pixels at x-coordinates and y-coordinates at the point (x, y). We then define operation g which takes in two integers and outputs another integer and define a new image H such that:

$$\forall x, y \in \mathbf{Z}\,, H(x)(y) = g(F(x)(y), G(x)(y))$$

Note that it's assumed we're only *really* caring about values x and y that fit in the domain of the image. Common operations that people might use are standard logical operations (AND, OR, XOR, etc...) for binary images, and standard integer operations for grayscale images (+, -, *, etc...)

3.6 Other Important Transformations

Here I am going to refer to some famous transformations which are called point-transformations – these are transformations that can be characterized as the output's particular pixel depending on the input's particular pixel (at that point):

$$G(m)(n) = f(F(m)(n)) \forall f \in \mathbf{Z} \to \mathbf{Z}$$

Some famous versions of some of these transformations include:

- **Digital Negative** - often stored in RAW files and retain a lot of the initial data in an image
- **Thresholding** - basically "sets" pixels to a particular value (usually maximum) if the pixel meets a certain intensity-threshold, and send the pixel to the minimum (usually 0) if the pixel doesn't meet the threshold.
- **Clipping** - does the same thing as thresholding but for a range of values instead of a particular value
- **Bit-Plane Slicing** - involves taking the grayscale intensity values and converting to binary representation; importantly, we produce multiple different binary images (0 or 1) for the rightmost bit, 2nd rightmost bit, 3rd ... etc
- **Contrast Sketching** - this point transformation uses certain parameters to highlight certain colors or intensities more than others
- **Logarithmic Transformation** - this uses a logarithmic function to drastically reduce certain colors (lighter ones normally). This makes it easier to see where high-intensity values are located relative to medium and low intensity.
- **Power-law Transformation** - this uses the gamma correction in order to further enhance certain pixels which brightens/darkens more sections accordingly
- **Others** - there are a LOT of others that I am not mentioning for simplicities sake, and also because these are things you know on a need-to-know basis (not really something you need to always know if that makes sense) – that being said, you can learn more about it the **College Friendly's Youtube Series on Digital Image Processing** in videos 6, 7, 8, and 9.
- **Histogram Equalization** - (NOT a point operation transformation, but another one I wanted to include) - this is a technique in which you assign a histogram to grayscale intensity values of an image. Using that, we end up constructing a PDF and CDF (probability distribution and cumulative distribution functions), and adjust the values of the pixels so they better fit per the average values of the buckets of the CDF. This is used to better enhance contrast and equalize some of the intensities of the image.

3.7 Linear Image Filters (Gaussian Filter)

Linear image filters are filters which can be applied to an image as a linear function on the pixels (or elements) of the image (viewed as a matrix). Importantly, it is equivalent to a convolution which is provable. I won't get into a proof, but there are many such proofs online. Let's first define the definition of two-dimensional convolution:

$$g(i, j) = \sum_{m=1}^{N} \sum_{n=1}^{N} f(m, n) h(i - m, j - n)$$

Note that for the case of an image, when applying a mask or a filter (ie a convolution with some type of discrete function of a certain size), dealing with the borders is significantly harder. So, what we end up doing is figuring ways around this – there are typically 3:

- Ignore border values (which just makes a smaller image because we need to ignore $\frac{M}{2}$ and $\frac{N}{2}$ amount of pixels on the borders.
- Pad with a constant value – we can take the average of the image and make the image big enough to accommodate convolving with the filter by padding with the average value
- This is the most common approach, but we can instead pad the outside of the image with the reflection of the inside of the image handling diagonals by averaging.

We can also observe the 2d convolution and note that it is basically a 1d convolution except with two sums, and a multi-argument function. We can progress from here to define the **impulse filter** which is simply a filter with one "dot" in the center of the "mask" (imagine the mask as a matrix which is being used to define a function convolved with the image, the matrix would have all 0's except one value of 1 in the center pixel). We can observe the **sifting property** stating that convolution with the impulse filter returns the initial image:

$$F * I = F$$

Let's also note two important properties regarding convolution (which have been mentioned earlier but are worth reiterating for the 2dimensional case):

- The "area" of the convolution or the amount of intensities in the entire mask must equal 1 or else the image will be oversaturated (which means the values, as aforementioned, will need to be clipped off at the highest intensity level allowed)
- The convolution will FLIP in BOTH directions (vertically and horizontally), the initial mask before applying it. So, if we define a mask H of length 4 by 4 with only entry $H(0)(0) = 1$ and all other entries non-zero, when we convolve with this max an image of something, the resulting image will be shifted UP and to the LEFT even though the "1" in the matrix would make you think we are grabbing values in the upper left and shifting them DOWN and to the RIGHT. *BE CAREFUL WITH FLIPPING!*

Now, let's try and define the notion of several very famous filters (**NOTE: kernel, filter, mask, and template all refer to the same thing**):

- **Box Filter** - this is one of the most famous filters (and was referenced earlier in the chapter) - it will, broadly speaking, simply grab an image and run a normalized box on it – it will simply "smoothen" the image (to me, it looks more like a blur).
- **Gaussian Filter** - this is a filter defined by the Gaussian function (normal distribution function for 2dimensions basically) that is often used for blurring or smoothening a picture - it is defined this way:

  $$n_\sigma(i,j) = \frac{1}{2\pi\sigma^2}e^{-\frac{1}{2}(\frac{i^2+j^2}{\sigma^2})}$$

  The $\frac{1}{2\pi\sigma^2}$ is for normalizing the function (in order to make sure that we don't oversaturate the image). That being said, it's also worth noting that σ is a parameter you can pick (it's essentially the equivalent to picking the standard deviation of a normal distribution). This will change *how* blurry the result becomes (or smooth depending on ur definition). Let's quickly reformulate the Gaussian kernel to see an important property (note that we are assuming the image is a square because we use K instead of M and N independently, but that shouldn't be too important):

  $$g(i,j) = \frac{1}{2\pi\sigma^2}\sum_{m=1}^{K}\sum_{n=1}^{K}e^{\frac{-1}{2}(\frac{m^2}{\sigma^2})}\sum_{n=1}^{K}e^{-\frac{1}{2}(\frac{n^2}{\sigma^2})}f(i-m,j-n)$$

We can see how the Gaussian filter can actually be observed as TWO convolutions together. This is a very important property of the Gaussian Kernel – it is separable. Thus, we can use a 1dimensional Gaussian filter and convolve it with the initial input, and convolve the result with a 1dimensional Gaussian filter in the other direction. Redefine the Gaussian filter V and H as the vertical and horizontal Gaussian filters and the Gaussian Kernel for 2 dimensions is essentially:

$$f * G = f * V * H$$

This reduces the complexity from $O(n^2)$ to $O(n)$ which makes computing the Gaussian blur/kernel really nice!

3.8 Non-Linear Image Filters (Bilateral Filter)

These are the set of filters which aren't linear shift invariant, and thus, can't really be implemented as convolutions. There are many useful non-linear kernels we would like to implement - let's start with the **median filter**:

The median filter is a filter which applies discrete convolution in some sense with a filter being applied over an image, but takes the median of the image. This is particularly useful for when there is a lot of noise in an image (particularly when it's fine or grainy noise) because things like the Gaussian kernel would instead smear the noise and make it more preeminent instead of actually getting rid of it. However, there's a better way to approach this problem of getting rid of the noise in an image without blurring up the whole thing: **the bilateral filter**.

Let's first visit the idea of the Gaussian filter. We want to only apply the Gaussian filter to pixels that are *similar* to the ones around this. So, we can introduce the idea of a Gaussian filter that *varies per pixel* in such a way that the filter is biased to include "less" of the pixels that are different from the original one (this way noise is ignored, and not smeared into the filter). Let's mathematically formulate this:

$$g(i,j) = \frac{1}{W_{sb}} \sum_m \sum_n f(m,n) n_{\sigma_s}(i-m, j-n) n_{\sigma_b}(f(m,n) - f(i,j))$$

There are **two** Gaussians in this formulation - one Gaussian is the spatial Gaussian which is used to "blur" the image out (so it just constructs the same thing based on distance to the pixel), but the other Gaussian is a brightness Gaussian function which is single-dimensional, but instead takes in the brightness, inputted as $f(m,n) - f(i,j)$ and outputs a coefficient based on that difference. If the difference in brightness is higher, then we end up with a a lower value, but if the difference is lower, we end up with a higher value. Also note that we have a normalizing factor in front to adjust the weight of the entire kernel (after adjusting for this) to make the energy/area equal to 1.

3.9 Template Matching and Correlation

This problem stems from trying to find a particular set of pixels from a filter (or a template as it's called per the namesake) in the image somewhere. It's important if you're trying to find some feature or object in an image somewhere. We can start by taking the template, and trying to compute the sum of squared differences between both of these in a matrix format:

$$e(i,j) = \sum_m \sum_n (f(m,n) - t(m-i, n-j))^2$$

This is basically computing an error function which returns a new matrix which determines the sum of squared error prevalent in the image. We can claim the pixel with the least squared error marks the region (or, in a more kernel-template approach, is the center of the region) where the template is located in the image. Let's expand upon this though:

$$e(i,j) = \sum_m \sum_n (f(m,n) - t(m-i, n-j))^2$$

$$e(i,j) = \sum_m \sum_n (f^2(m,n) + t^2(m-i, n-i) - 2f(m,n)t(m-i,n-j))$$

Since $f^2(m,n)$ and $t^2(m-i,n-i)$ are already decided and we can't *change* this in any form, let's instead focus on the last term. If we want to find where the error is as low as possible, and there is a subtraction of $2f(m,n)t(m-i,n-j)$ then we can claim that reducing the error is equivalent to maximizing that quantity! We can note (and this isn't really a coincidence), that the term $f(m,n)t(m-i,n-j)$ inside a summation looks *a lot* like convolution. However, this is defined as **cross-correlation**:

$$R_{tf}(i,j) = \sum_m \sum_n f(m,n)t(m-i,n-j) = t \star f$$

This is very similar to discrete convolution, except (with a huge difference), we don't FLIP the kernel/template. This obviously makes sense because if we want to find the similarity between two things, we don't want to alter the templates structure in any way. Thus, in some sense, convolution and cross-correlation are very similar (one just flips the kernel both ways and performs dot-wise multiplication, and the other just omits the flipping). We do need to account for the fact that, as mentioned earlier with oversaturation the energy of the kernel must be 1 – however, this isn't necessarily going to happen because of how we are trying to find something else. So, we need to introduce the **normalized cross correlation** which solves the issue of normalization by dividing through the correct factors:

$$N_{tf}(i,j) = \frac{\sum_m nf(m,n)t(m-i,n-j)}{\sqrt{\sum_m \sum_n f^2(m,n)}\sqrt{\sum_m \sum_n t^2(m-i,n-j)}}$$

I won't get too deep into this, but Shree Nayar's video on Template Matching (First Principles of Computer Vision) is a great resource for this!

3.10 Image Filtering in Frequency Domain

Since I have already written the section on Fourier-Based Analysis, I don't want to get too far into this. Broadly speaking, this whole idea just takes the main ideas of low-pass filters, high-pass filters, and the Fourier transform, and extends them into two dimensions. As mentioned in the end of section 1.2, convolution and multiplication have a special relationship in the context of the interactions between the spatial and frequency domains – that being said, let's restate the fact (the premises of the fact are given first):

$$G(u) = \mathcal{F}(g(u))$$

$$H(u) = \mathcal{F}(h(u))$$

$$G(u)H(u) = g(u) * h(u)$$

Note that the $*$ operator is referring to **CONVOLUTION** not multiplication (or else this is patently untrue). We can also note that as a result of the Fast Fourier Transform (mentioned in Section 2.2 using the Cooley-Turkey approach and the Danielson-Lanczos lemma), computing the Fourier Transform and it's inverse is extremely cheap. Thus, instead of computing the convolution of two functions, it is *very* common to instead use the Fast Fourier Transforms to transform both functions $g(u)$ and $h(u)$, and multiply the results, and perform the Inverse Fast Fourier Transform (which is almost the same algorithm as the Fast Fourier Transform) in order to obtain the same result as convolving. This simplifies computation and actually reduces runtime from $O(n^2)$ to $O(nlgn)$. This is especially the case for the Gaussian kernel. Note that this isn't necessarily applicable to the Bilateral filter because it is NON-LINEAR and can't be represented as a convolution.

3.11 Deconvolution

This is the idea of taking an image that has been convolved and deconvolving it to get the initial signal. This is particularly useful for places where certain convolutions have taken place that are unwanted and we want to remove them. Take for example motion-blur that happens because a camera was moved during the taking of a photo – it can be represented as a convolution.

$$P * M = F$$

M represents the Motion Blur as a Convolution, P represents the initial photo, and F represents the final photo outputted. We can determine M by using the gyrosensor on the phone to find how the phone was handled during the taking of the photo. We have the final result G as well, but we want to get the initial P to get the photo without the motion blur – let's use the Fourier Transform and the theorem regarding convolution to help out:

$$P * M = G$$

$$\mathcal{F}(P)\mathcal{F}M = \mathcal{F}G$$

Thus, we can state that the initial photo P must be defined as:

$$P = \mathcal{F}^{-1}(\frac{\mathcal{F}G}{\mathcal{F}M})$$

This is a great approach to getting the initial image back – however this is a bit *too* ideal because it annoys any notion of noise that can occur. Note that there is going to be additional noise introduced during the motion-blur because the image is changing while it's being captured with a shaky camera (and in general, noise from the image sensor)! Let's add some noise to the problem by noting the following where μ refers to noise:

$$P * M + \mu = G$$

Unfortuantely, the approach above simply doesn't work because applying the Fourier Transform with all this noise creates *too* much noise (the signal is overwhelemed by noise). This is actually the case for two principal reasons:

1. When $M = 0$, we end up getting an undefined result plugged in the Fourier Transform which makes it unrecoverable. Note that it's not unreasonable for the Fourier Transform at a particular point to be 0 (note that I'm choosing to avoid indexing, but it's assumed M is a matrix and so it can be indexed with $M(m)(n)$).
2. The motion-blur filter we assign to at the start is a low-pass filter on its own. This is becuase the motion-blur filter, broadly speaking, is getting rid of the higher frequencies, or the ones that contribute a lot more to the specifics of the image. This means that broadly speaking, it is actually *VERY* likely for it to be 0 at certain points.

Both of these properties really screw with adding TOO much noise in the image (our image will end up looking like a '70s TV when there's nothing on it). We can define a new type of deconvolution to handle this called **Weiner Deconvolution**. I won't really get into this because it's a bit too mathematical and not *necessary*, but if you are interested please check out Shree Nayar's 12th video in the Image Processing playlist around the 8-9 minute mark for more information on how we can introduce a different type of deconvolution to deal with the added noises in the image, and ideally rid it of some of the motion blur. Do note that no solution is perfect, so the ultimate result will have some added noise somewhere in the result no matter how much we try to prevent that, but broadly speaking, it's effective at it's job and produces a more clear image!

3.12 Further Learning

There are many good resources to learn more about this, but I'll only list the ones that I used and found particularly helpful.

- **First Principles of Computer Vision's series of Image Processing** - this is an oustanding lecture series. A lot of the content I mention including formulas and intuition is from this series. Please watch it!
- **College Friendly's Digital Image Processing Series** - another stellar lecture series. It includes more information beyond image processing, and also goes a bit too in-depth (for purposes of this notes) in things like examples and exact methods, but it's also a great place to look at.
- **Szeliski's Computer Vision and Algorithms Texbook** - this is a great textbook, but I didn't really use the sections in this textbook for this. Definitely use this textbook if you're interested in computer vision though!

4 Edge Detection

Note that Edge Detection is one of the most important yet oldest topics in computer vision so I will be discussing many different styles of implementation for this. Normally edge detection is done in order to find edges, but more importantly to export the edges as a feature for various machine learning models. You may find it pretty intuitive - if we want a model to perform **image segmentation** and **object recognition** (check section 3.2 if you forgot this), it makes sense that we should export the edges of an image as a feature so the model can try and discern what lies "within" edges, what lies "outside" of edges, and which edges are used to characterize noise and unimportant things (shadows, background noise, etc...)

4.1 Edge Detection using the Gradient

Let's first start this section by acknowledging what the gradient is. Assume a multivalued function $f(x, y)$, we define the gradient as:

$$\nabla f = \begin{bmatrix} f_x & f_y \end{bmatrix}^T$$

Note that indices indicate partials in this case. We can also define the notion of magnitude (S) and orientation (θ) of the gradient vector by saying:

$$S = |\nabla f| = \sqrt{(f_x)^2 + (f_y)^2}$$

$$\theta = \arctan(\frac{f_y}{f_x})$$

Note that the θ being defined the way it is is a product of how we define the direction of the gradient in Multivariable Calculus. That being said, we can also note that if the magnitude of the gradient is *large*, that means the $f(x, y)$ changes significantly in the nearby region. Let's try to discretize this in order to talk about applying this as an image, and do this by constructing a 2x2 filter/convolution matrix f instead of a multivariate function noting that ϵ is the distance between the two pixels:

$$f = \begin{bmatrix} f_{i,j+1} & f_{i+1,j+1} \\ f_{i,j} & f_{i+1,j} \end{bmatrix}$$

$$f_x \approx \frac{1}{2\epsilon}((f_{i+1,j+1} - f_{i,j+1}) + (f_{i+1,j} - f_{i,j}))$$

$$f_y \approx \frac{1}{2\epsilon}((f_{i+1,j+1} - f_{i+1,j}) + (f_{i,j+1} - f_{i,j}))$$

Note that these are basically just using discrete equivalences to the partial derivative. For the f_x case, we are simply computing the difference of the right column and left column in both rows (effectively seeing how much columns change going forward in the x-direction). The same thing is true for the f_y case. The fraction $\frac{1}{2\epsilon}$ is because we are just averaging the differences of the intensities of the pixels (and since there are two, we need the fraction $\frac{1}{2}$).

We can actually, as you may have guessed, implement this as convolution. Note that because convolution flips prior to computing the "dot product" or pairwise summation between the filter and image, we need to flip the filters for f_x and f_y in both the vertical and horizontal directions (note for this case that $\epsilon = 1$)

$$f_x = \frac{1}{2\epsilon}\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

$$f_y = \frac{1}{2\epsilon}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$$

First, note how this makes a lot of sense (I won't go into this, but the math behind that matrix should show how we're checking the differences in the X and Y directions). Second, let's discuss some famous well known gradient-based operators based on the general idea I introduced above:

- **Roberts Filter** - this is a 2x2 filter which is somewhat based on the gradient but has more of a focus on diagonals - note how changes in the positive-45-degree angle are denoted as changes in X while changes in the negative-45-degree angle are denoted as changes in Y:

$$f_x = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \; f_y = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

- **Prewitt Filter** - this is a 3x3 filter which is almost just a 3-d analogue of the gradient filter initially introduced:

$$f_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \; f_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

- **Sobel Filter** - this is very similar to the filter mentioned above (and can scale up to 5x5 as well but I won't go into that) - this particular filter, as you will notice, will focus a lot more on horizontal and vertical edges as opposed to diagonal ones because of where the 2 is placed:

$$f_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \; f_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

For some of the larger filters, sometimes we might convolve the filter with the Gaussian in order to have a more smoothing effect (to make things less susceptible to noise). Speaking about that, let's talk about limitations of the aforementioned gradient-based filters. We have a few main concerns:

- **Localization** - the smaller filters are better for localization while the larger ones aren't and this follows naturally because the smaller filters check a very small range of pixels. This means that noise in the surrounding area of a pixel would impact the larger filters but not the smaller ones.
- **Noise Sensitivity** - both large and small filters are susceptible to noise, but naturally, smaller ones are significantly more sensitive because they're analyzing a smaller region.
- **Detection** - broadly speaking, the larger ones are better for edge detection because edges tend to be "big" in many real-life images.

Let's discuss how to use the filter. When convolving this with every single pixel, we will get a "number" (which represents the derivative in both directions). To get some more context, the "gradient-based filters" are checking the magnitude of "change" in individual directions. In order to perform further analysis, we compute the total magnitude of the gradient noting $|\nabla f| = \sqrt{f_x^2 + f_y^2}$. However, almost every single pixel is going to return some type of value, so how do we pick the ones that form an edge versus the ones that are "small" enough to not really be considered one... that brings us to thresholding!

- **Standard Thresholding** - this is basically picking an arbitrary T as a threshold and making something an edge accordingly:

$$|\nabla f(x,y)| < T \rightarrow \text{NOT EDGE}$$

$$|\nabla f(x,y)| \geq T \rightarrow \text{EDGE}$$

- **Hysteresis Based Thresholding** - hysteresis basically means checking the surrounding region as well to determine something. For this, it means we use two thresholds T_0 and T_1

$$|\nabla f(x,y)| < T_0 \rightarrow \text{NOT EDGE}$$

$$T_0 \leq |\nabla f(x,y)| < T_1 \rightarrow \text{EDGE if 4 or 8-adjacent pixel is DEFINITELY an edge}$$

$$|\nabla f(x,y)| \geq T_1 \rightarrow \text{ DEFINITELY AN EDGE}$$

Note that 4-adjacency and 8-adjacency is defined earlier in Section 3.4 as being the surrounding neighboring pixels.

Let's talk about a practical implementation. Practically speaking, we would compute the magnitude of the gradient by using both the f_x and f_y filter and performing the pointwise operation of squaring each pixel, adding, and taking the square root. Then, we can threshold each value by whichever thresholding scheme we choose to implement, and go further from there. If you would like to see an implementation of the Sobel Filter, **check my repo**.

4.2 Edge Detection using the Laplacian

Warning: this can get a lot more mathy, but for sake of discussing the idea, I will omit a lot of the math - please watch Shree K Nayar's video in the First Principles of Computer Vision available *HERE*

Let's now define the notion of a Laplacian. For those who recall from multivariable calculus, this is:

$$\nabla^2 f = (f_x)^2 + (f_y)^2$$

Note that in a very pure sense, an edge is a line across which we observe a rapid change in the intensity values of a pixel. This means that we notice some type of positive derivative. In a second derivative sense, this is denoted by a zero crossing because the growth of the peaks during the first part of the increase, and then falls during the second part of the increase.
We can use the previous notions of a discrete gradient or derivative to now define the Laplacian Filter by taking the filters of f_x and f_y and approximating a new discrete operator! I won't go through this, but it isn't particularly challenging. The main thing to note, however, is that we need a minimum of a 3x3 matrix instead of a 2x2 matrix (this should be relatively obvious, but we need to have 4 2x2 matrices located somehow in the matrix to be able to compute a 2nd derivative). There are some modifications we do, but a standard version of the matrix we end up getting is:

$$\nabla^2 \approx \frac{1}{6\epsilon^2}\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

You should get the broad paradigm shift now that instead of trying to find the change between the left column and the right column, we're comparing the left and right columns with the center (and vice versa for rows). Here are some considerations for how we run the Laplacian:

- **Negative Values** - note that the Laplacian, more particularly for non-edges, will return values that are actually *negative*. We deal with this by instead mapping the values from $[-128, 128]$ to $[0, 256]$.

- **Zero Crossings** - note that the zero-crossings idea is then used to determine, from the initial Laplace filter, which of the components are edges. This is done by checking where rapid changes across the x-axis (or in this case, across any edge) take place, and marking those as edges.

This is hard to explain without graphics, so check out my repository and Shree K Nayar's video to learn some more.

4.3 Gaussian Derivative and Laplacian of Gaussian Filters

Here is where we introduce an idea that was briefly mentioned earlier, which is Gaussian Convolution. The issue with trying to detect edges is that we, are purposefully trying to find changes in the input signal, and then classify those as edges. However, noise can also be characterized as changes in the input signal. That raises a dilemma because when we run a Gradient or Laplacian based filter on the input data, we end up with a huge mess because all the noise also significantly contributes to the derivative.

We deal with this in various ways, but let's start by addressing the first – running the Gradient and Laplacian-based filters on the Gaussian Kernel of the image. This smooths the image first, and *then* adds on the edge detection. While this screws slightly with edges because it smoothens out, so any sharp edge becomes slightly smoother and relatively less "obvious" edges become harder to detect, it rids the impact of noise. Let's formulate this:

$$\nabla(n_\sigma * f)$$

However, convolution is associative, and the ∇ operator is nothing but a convolution, and so, we can claim the following:

$$\nabla(n_\sigma * f) = \nabla(n_\sigma) * f$$

This means we can simply compute the ∇ of the Gaussian and apply that as a convolution. This would make the convolution one step (instead of 2) because the magnitude of the gradient of the Gaussian isn't particularly challenging to compute. We can then simply continue with the thresholding process and determine edges that way without worrying about noise. Similar logic applies to the Laplacian:

$$\nabla^2(n_\sigma * f) = \nabla^2(n_\sigma) * f$$

This ultimately makes a single-operation filter. When done with a Gradient-based Detector, we call the final single-operation convolution **Gaussian Derivative** (or Gradient of Gaussian) filter. When done with the Laplacian filter, we term it the **Laplacian of Gaussian** filter. Let's start by comparing the two:

- **Location, Magnitude, and Direction** - the Gradient based edge-detector gives you location, magnitude, and direction while the Laplacian only provides location of the edge (because Gradient is a vector)
- **Detection** - the Gradient based edge-detector is thresholded using either standard or hysteresis based thresholding. On the other hand, the Laplacian relies on the zero-crossings – you do need some type of way to measure the "steepeness" of the zero-crossing to verify if it is actually an edge, but this isn't as rigorous as thresholding.

Now, these edge detection schemes work great, but let's try to combine them to make something that works even better by harnessing the strengths of each!

4.4 Canny Edge Detector

This is arguably the best and most widely used edge detector in Computer Vision. This basically combines all the filters mentioned above in some capacity: the Gaussian Kernel, the Gradient Kernel, and the Laplacian Kernel. Let's walk a bit through the steps of the Canny Edge Detector and briefly mention why it's valuable! Note that I refers to the input image and F refers to the output.

1. We run the 2D Gaussian to smooth out the image. We can pick σ based on how much we want to smooth out the image (standard deviation of the normal distribution).

$$F = n_\sigma * I$$

2. We compute the Image Gradient using some type of Sobel Operator (usually 5x5). This is the equivalent of the **Gaussian Derivative** filter.

$$F = \nabla(n_\sigma) * I$$

3. We compute the magnitude of the x and y directions using Euclidean norm:

$$|\nabla(n_\sigma) * I|$$

4. We then find the gradient orientation at each pixel by constructing a normalized (unit) vector in the direction of where the edge *should* be located at that point.

$$\hat{n} = \frac{\nabla(n_\sigma) * I}{|\nabla(n_\sigma) * I|}$$

5. We now compute a one-dimensional Laplacian (which is *basically* like the 2nd derivative) along the direction of the unit normal vector
6. We check if there is a zero-crossing in that direction, and if so, claim that pixel must be an edge.

This is particularly helpful because it uses both techniques to find the likely places where there are images, and doesn't allow noise in other regions to corrupt the edge detection algorithm as much because it uses the Laplacian filter *only* in the direction of the edge. Check out my repo online for an implementation of this!

4.5 Corner Detection

Corners are defined as a point where two edges in an image intersect. We need to use the derivatives of the image (previous filters mentioned) in order to find where corners are located. We first start this task by thinking of different types of images:

- **NO EDGES** - such an image would have no real "edges" anywhere (basically a unicolored image), and so, we can say that the set of image gradients if potted as a vector (S_x and S_y) would ultimately yield a circle very close to the origin because their magnitudes would be relatively small.
- **ONE EDGE** - there is no corner in this image, but the one edge means that there would be high gradients going in one particular direction. This would mean a large circle-like cluster near the origin (as mentioned above), but *also* a collection of gradient vectors that lie along one particular line because there is a general change in a given direction.
- **TWO EDGE / CORNER** - this would normally lead to some ellipsoid/circle looking figure where we get the cluster near the origin, but two lines protruding from the origin going out in relatively orthogonal directions (otherwise the corner is more like a "fat edge" if that makes sense)

From this, we fit an ellipse onto the collection of vectors – this is done through finding the largest moments in the data – this has a very clear correlation to Principal Components Analysis where we're identifying the vectors where there are the highest amounts of change. Let us now denote the ellipse with semimajor axis length λ_1 and semiminor axis length λ_2. We can classify no edge, edge, and corner using this:

- **FLAT:** $\lambda_1 \approx\approx \lambda_2$ and both λ_1 and λ_2 are relatively small
- **EDGE:** $\lambda_2 >> \lambda_1$ or $\lambda_1 >> \lambda_2$ means there is an edge
- **FLAT:** $\lambda_1 \approx\approx \lambda_2$ and both λ_1 and λ_2 are significantly large

We can also introduce the idea of the **Harris Corner Detection** which is the above idea formalized into a "is this a corner or not?" dynamic using the equation:

$$\lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

We can define some type of threshold R and if the value from above is less, then it's not a corner, and otherwise, is. However, a common issue we run into is having *too many* peaks. We want singular points to be corners, not a whole cluster of points, and that topic lends nicely to the idea of **non maximal suppression** through the following steps:

1. We slide a window of size k over the image, and check all the pixel values in that window.
2. If there are any values that are greater than the one we are currently checking, we set the current one to 0.
3. If not, we just move on.

Note how this basically just only lets the "maximums" in a general area (window of size k) remain, and *suppresses the non-maximums*!

4.6 Boundary Detection

Boundary detection is very similar to edge detction, but instead really hones in on trying to connect the parts of the image to make more sense of what's in the image. For example, boundary detection would try and get rid of the unnecessary noise you would get after performing edge detection (like the noise you would see in my GitHub Repo for any of the Edge Detection algorithms). Let's first talk about the steps we would take from an image to get to boundary detection:

1. **Edge Detection** - this is detecting the edges using some type of filter such as the Canny filter - this is a natural first step
2. **Thresholding** - here we try to apply the thresholding procedure to get rid of *some* of the noise and amplify the parts of the image we believe are actually edges.
3. **Shrink and Expand** - we run a binary image processing technique in order to further get rid of noise
4. **Thinning** - since thresholding assigns a binary value to each edge, it often makes the edges appear more *bold* than they should be so we use thinning to try and make the image less bold and reduce the width of some of the edges
5. **Boundary Detection** - it's from here where we need to use boundary detection to go further in the edge detection algorithm so let's get to work.

This problem basically boils down to least-squares and a lot of linear algebra trying to decide which "formulation" of lines or curves is best for the edges we have. Let's first start by acknowledging what it's like for the case of two points when we're trying to fit a line of best fit between them:

$$y_0 = ax_0 + b$$

$$y_1 = ax_1 + b$$

Note that we can reformulate this as:

$$\mathbf{X} = \begin{bmatrix} x_0 & 1 \\ x_1 & 1 \end{bmatrix}$$

$$\mathbf{Y} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

$$\mathbf{X} \begin{bmatrix} A \\ B \end{bmatrix} = \mathbf{Y}$$

Note that although for this case since we have 2 datapoints and 2 results, the result is pretty easy to solve (as in, so long as x_0 and x_1 are different points, the system is invertible so we can just take the inverse of the square matrix and determine the coefficients), this isn't likely always going to be the acse. More times than not, the system will be overdetermined with less unknowns than observations (or edges), and so, we need to note the following technique to solve this:

$$Ax = B$$

$$A^T Ax = A^T B$$

$$(A^T A)^{-1} A^T Ax = (A^T A)^{-1} A^T B$$

$$x = (A^T A)^{-1} A^T B$$

Note the key observation that $A^T A$ is guaranteed to be a square, invertible (symmetric) matrix. This is essentially performing a column-space projection on B, and then solving the system. Note that if we want to incorporate a higher degree of polynomials, we can use the same approach by defining a wider matrix and having more coefficients. This is great because now we can always get "curve" or "line" that best fits where the edges are located, and have performed one of the first tasks of boundary detection!

4.7 Active Contours

Let's define an **active contour** as taking a generic random (or human-drawn) countour and iteratively deforming it to match a boundary of an image. Think of drawing a random circle around a coin on a blank background, and trying to iteratively get the circle to deform to the boundary of the coin. This is particularly useful for videos (because it can track object movement), and furthermore, for medical analysis such as analyzing scans and things of that nature. Now, you should be noticing something – this is exactly the same idea as "backpropagation" or otherwise, **gradient descent**.

Let's force our contour to go in the direction of the of the maximum gradient, and thus, maximize the sum of the gradient magnitude square (because we want to go in the direction of the most change):

$$\text{We want to maximize gradient which is } |\nabla I|^2$$

Let's also note that convolving the gradient with the Gaussian kernel (blurring up the image), is going to "smear" the gradient around a bit that way the gradient is more present in the image. This is particularly important in cases where the initial contour is located in an are which has few edges around it; we want to blur the edges so they can “reach” the area of the initial contour! Note that $I(v_i)$ refers to a function of the vector $< x, y >$ where if the image is $f(x, y)$, $I(< x, y >) = f(x, y)$ – basically, this function I takes in a vector of the x and y coordinates and outputs the intensity value. This is what we want to effectively maximize:

$$\sum_{i=0}^{n-1} |\nabla n_\sigma * I(v_i)|^2$$

Thus, we also want to minimize the negation (helps a bit later down the line):

$$E_{image} = -\sum_{i=0}^{n-1} |\nabla n_\sigma * I(v_i)|^2$$

We call E_{image} as the image term. Now, we're also going to try and make the contours **elastic** and **smooth** (basically, we want the contour to behave like a rubber-band). And in order to do this, we need introduce the idea of **Internal Bending Energy**. This is a physics notion which I will simply define and not explain:

$$\text{The elastic energy of the contour is } E_{countour} = \alpha E_{elastic} + \beta E_{smooth}$$

$$E_{elastic} = |\frac{d\vec{v}}{ds}|^2$$

$$E_{smooth} = |\frac{d^2\vec{v}}{ds^2}|^2$$

Note that elasticity is the speed and smoothness is the acceleration in terms of multivariate calculus. We can discretize these noting the definitions of discrete change and derivatives, and use them both to make a new term called the **total energy** of the contour:

$$E_{total} = E_{image} + E_{contour}$$

Now, let's try to define a greedy algorithm to use to iteratively change the contour to match the contour of a boundary!

1. **Sample** - we first need to (obviously) sample where the contour is located to get certain points - we can pick the sampling based on how the contour looks, and how much noise the image has.
2. **Adjustment** - we can adjust v_i to a position within a window W (same idea as a discrete convolution) where the function E_{total} is minimized.
3. **Stop or Repeat** - if we reach a point where the sum of all energy/motions of the contour points is less than a particular threshold, we stop, or else we continue the process again.

This is really good! We now have an algorithm to iteratively go from a contour and shrink it to the actual boundary of an object! Let's discuss some complications!

- **Multiple Boundaries** - note that since we're also trying to minimize the *internal bending energy*, we are forcing our contour to behave like a rubber-band. We can adjust α and β to determine how much like a rubber-band the contour behaves.
- **Precision Limitation** - also note that the blurring I referred to the beginning is only so effective – the initial contour needs to be placed in the right general location near some type of boundary or else it won't detect which direction to go to in order to get the boundary where it should be.

- **Other way around?** – if we, instead of computing the minimum total energy in the window, compute the maximum total energy, we would be reversing the effect and finding the edge on the *other side* of the contour!
- **Additional Constraints** - we can add more constraints in the *total energy* component if that's something you think would fit well for your image. If you know your image, for example, is going to be looking a certain way or behaving a certain way, you can try and force the active-contour algorithm to use that to make the result look more like what you want it to look like.

4.8 Hough Transform

The Hough transform notes a simple technique to change lines to points and points to lines. It's particularly useful for finding a line in a set of various different points. Let's first start by noting that a line can be described as:

$$y = mx + c$$

However, we can reformulate this to become:

$$c = mx - y$$

This allows us to go from the **image space** (of x-y coordinates in an image) to the **parameter space** (possible parameters that could connect the various x-y coordinates. You may notice that an (x, y) input in the parameter space corresponds to a *set of lines* characterized by $y = mx + b$ potted on the (m, c) parameter space. The Hough Transform utilizes this fact to replace plotting coordinate-vectors in the (x, y) space with (m, c) in the parameter space that represent all lines passing through (x, y), and so, if we repeat this process with multiple (x, y) points, the intersection of those produces a single (m, c) point that represents a single line (not point).

We can now introduce the idea of an **accumulator array** which transforms the matrix representing an image to a matrix of the parameter space. For every single edge represented as a point in the image space, we augment the corresponding parts of the parameter space associated with the edge in the image space by 1. We obviously first need to initialize the array to all 0's. We then pick the parameter-coordinate representing the highest numerical value in the array.

For sake of easing the math I won't go into this, but it's very common to use the below formulation instead: $x \sin\theta - y \cos\theta + \rho = 0$ This avoids some of the issues of the slope being "too" large (which represents an issue if the line we want is supposed to be vertical). We can use a polar-based approach (ρ, θ) in order to have less such issues. Note that instead of mapping points to lines, it instead maps points to sinusoids/curves. This is also something that is very easily transform-able to circles or other lines/curves depictable with two parameters. Let's now introduce the **General Hough Transform** to find images that may not easily be able to be depicted with 2 parameters, but instead by a model. Let's first assume we get some type of "model" which refers to an image which we want to find in an image map. We then need to construct the "Hough Model" for the associated image:

1. **Edge Map the Model** - if we're given a model image to use to base the Hough Model on, it is imperative to ensure that we are also checking hte
2. **Pick a reference point** - we need to use vectors to construct the Hough model, and so, need a reference point to base the "tails" of the vectors. This reference point is typically somewhere in the center of the object.

3. **Mark all vectors going in the same direction** - for all vectors that are going in the same direction, we construct a ϕ table. All vectors with the same ϕ (denoted as angle with the horizontal) are grouped together.
4. **Create an accumulator array** - this array is a direct copy (almost) of the image we're scanning for, except with different resolutions depending on what specifically we want. The goal is to iterate and go through the accumulator array, and count the number of times the surrounding vectors constructed from the edges match the results of the phi table
5. **Analyze the maximums of the accumulator array** - we now need to find the local maximum of the accumulator array to make any sense of the

Now, what does this mean? This means that if we run the General Hough Transform with a certain model of a tree (for example) on an image named B. C which is the output of the GHT on B will result in an image which is colored/shaded in such a way which indicates with likelihood how certain we are points are the *center* or *reference point* of that particular object. This means if we pick the reference point to be the root of a tree and we place multiple of those trees in an image, all the roots of those trees will be shaded or denoted as being possible reference points.

4.9 Scale-Invariant Feature Transforms (SIFT) Detectors

We need to find a way to "find" something in another image. You might recall from image processing (Section 3.9), the idea of Template Matching and Correlation which we can use to find another "template" in a broader "image". However, we note several important problems with generalizing that idea:

- **Image rotated** - if the template is rotated within the image, it would screw with the correlation metric and template-matching scheme
- **Image zoom** - if the image is zoomed in or zoomed out too much, it would also screw with the correlation metric (because the individual pixel values would be off).
- **Image complication** - if the image is too complicated or just has too many details, small changes in that may affect how accurate the correlation metric is.
- **Image occlusion** - by far the hardest thing to deal with, what happens if the image is blocked up in certain areas? That would *definitely* screw up with a lot of the template matching and correlation metrics I spoke about earlier.

One possible, but very *crazy* solution is to make many small templates from the initial image, and have them rotated and zoomed in at different points, and compute where the individual templates are located in the big image. *However, that is stupid and computationally intensive, and not to mention, stupid.* Let's introduce the idea of the **SIFT detectors** as a better approach for finding templates in an image! Let's note some important truths about finding templates in an image first:

- **Blobs as Interest Points** - we need to analyze "blobs" in an image, we can't simply be analyzing corners, edges, lines, or pixels. Otherwise, this just becomes *way* too susceptible to noise, and practically indiscernible. Such a blob needs to have a particular **location**, **size**, **size**, **orientation** , and **description**.

We'll talk about detecting blobs in the next section. Note that the term "Scale Invariant Feature Transform" is a direct extension of the idea of being able to detect certain features irrespective of appearance and scale (size and angle).

4.10 SIFT Detector: Detecting Blobs

Note that when I refer to "extrema", "minima", and "maxima", they are all in reference to basically the same thing which is maximum distance from the origin – these distances are positive, but the displacement/y-coordinates are negative, so keep that in mind.

This particular topic is challenging to understand, so I want to be careful. Let's first start by trying to find "blobs" in a 1-dimensional signal. There is more structure to how this is performed, and you can learn this in video 14 of the Edge Detection Youtube Series from Shree K Nayar. Let's start by defining the input as $f(x)$ and proceed through the following steps:

$$\text{Signal: } f(x)$$

$$\text{Normal Distribution: } n_\sigma$$

$$\text{Laplacian of Gaussian:} \frac{\partial^2 n_\sigma}{\partial x^2}$$

$$\text{Convolve with Signal:} \frac{\partial^2 n_\sigma}{\partial x^2} * f(x)$$

Understand that the second derivative of the normal distribution divides a factor of σ^2 which makes the peaks of the signal significantly smaller. In a traditional analysis of the second derivative we're looking for zero-crossings, so this doesn't really matter. But, for analyzing "blobs", it does matter, so we use a variant of the LoG for single-dimensional signals where we normalize by σ: it is termed, "**sigma-normalized 2nd derivative of Gaussian**":

$$\sigma^2 \frac{\partial^2 n_\sigma}{\partial x^2} * f(x)$$

We want to find the σ which maximizes the distance from origin of the above function because this finds the "width" of the blob. The mechanics of how this works won't be explained, but understand that we're choosing the parameter to make the normal distribution (second step from above) roughly match the width of the signal, and when we perform the above process, a general truth is that when such an alignment between the normal distribution and signal are obtained, the distance from the origin is maximized.

The conclusion is that local extrema in the (x, σ)-Space represent blobs. This means that if we imagine the sigma-normalized 2nd derivative of Gaussian defined as a function of x and σ, the extrema (more particularly the minima) represent blobs. This idea can be really nicely extended to 2 dimensions!

$$\text{Laplacian} = \nabla^2 f = (f_x)^2 + (f_y)^2$$

$$\text{Normalized Laplacian of Gaussian: } = \sigma^2 \nabla^2 n_\sigma$$

The **Normalized Laplacian of Gaussian** or **NLoG** is what we use for blob detection in two dimensions, and performs a transformation from the image space to the **scale space** which is an extension of (x, σ) to (xy, σ). We can then treat applying the NLoG as iteratively computing the maximum of the argument of the same function for different levels of the Gaussian kernel of the image. Imagine taking the images ordered from $(\sigma_0, \sigma_1, ..., \sigma_k)$ where k represents the scale (usually goes until the width of the image), and then computing the peak/max-argument of the NLoG on each of these images, and finding the maximum between all of them. This would tell me at a given (x, y) point, the size in radius of a circle at that point, σ, of a given feature located there. This feature could be, broadly, any blob or detectable "difference" in the image at that point. If we can't find any conclusive maximum of the NLoG at any particular σ, this would tell us that there is no particular feature at that given (x, y) point.

4.11 SIFT Detector (DoG and HoOG)

Difference of Gaussians (DoG) and **Histogram of Oriented Gradients (HoOG)** are techniques we use to augment the basic idea of a SIFT detector which I mentioned earlier. Let's define them:

- **Difference of Gaussians** - the following idea will not be proved, but is used for computing the Laplacian of Gaussian faster:

  $$\text{DoG} = n_{s\sigma} - n_{\sigma} \approx (s-1)\sigma^2\nabla^2 n_{\sigma}$$

  We can now notice that instead of computing the Laplacian of Gaussian at differing σ by just picking different multiples, we can instead imagine taking some image denoted F, and computing a tensor (vector of vectors) denoted as a large vector S which is composed of the image with various amounts of the Gaussian kernel applied on it in multiples of the σ – first σ, then $s\sigma$, then $s^2\sigma$, all the way until $s^k\sigma$. Importantly note that we can take each of these various Gaussian kernels at different σ and compute the differences of them in order to get "approximate" Normalized Laplacians of Gaussians (NLoGs) off by some sort of factor of $(s-1)$ as mentioned by the above idea.

- **Histogram of Oriented Gradients** - this broadly refers to computing the **angles** (of the gradient) of some image in a discrete region, and placing them on some sort of histogram. The goal is to compute the **principal orientation** by finding the most common angle. This would denote the "angle" that a feature from an input image is presented as in the resulting image.

4.12 SIFT Detector - Descriptor and Full Process

Let's use the techniques we mentioned earlier to make the entire SIFT Detector:

1. **Extract Interest Points** - We take an image in its discrete-sampled form. Note that this means the image can be in RGB format, and more importantly, it isn't necessary for an edge-detection filter to have been applied to the image.
2. **Difference of Gaussians 1** - We then apply Gaussian filters with different σ's based on a factor of s – it isn't important what the factor is, just that we continue from s^0 to s^k.
3. **Difference of Gaussians 2** - We then use the Difference of Gaussians approximation for NLoG and obtain $k-1$ different outputs which represent the Normalized Laplacian of Gaussians at each individual σ.
4. **Extrema** - Recall that now we want to compute where the extrema (particularly the maxima because we care for argument/distance from origin) are in order to find the general "radii" of certain features. So now we run a 3x3x3 extremum filter (if it's the maximum, then keep the value, or else, make it 0).
5. **Threshold** - Remove some of the weak extrema by creating a threshold on certain radii that meet a distance criteria.

Well, what do we do from here? Let's first start by trying to summarize SIFT Features in a meaningful way. We first need to acknowledge a couple of things:

- We want scale-invariance (thus, we need to be careful about adjusting with Histogram of Oriented Gradients)
- We want shift-invariance (thus, we need to normalize images with respect to the σ used in the Difference of Gaussians plane where the feature is detected).

- We do **NOT** want lighting or things of the sort to impact the general description of the features. This means using the magnitude of the gradient is not a great approach.

So, in order to take a SIFT feature and describe it in a good way, we use histograms of the orientation of the gradients of the feature. This means that we take a general feature (which is represented as a "blob" in an image represented by a circle of some given radius r), and then place all the angles of the gradient in a histogram. We can use many different methods in order to compare the different SIFT features. Assume for this case the histogram is represented by two arrays of data of length N, $H_1(k)$ and $H_2(k)$.

- **L2 Distance**
$$d(H_1, H_2) = \sqrt{\sum_k (H_1(k) - H_2(k))^2}$$
This basically measures the distance between the two vectors – when it's 0, we have a perfect match.

- **Normalized Correlation** - the formula for this can be found earlier, but this is a 2 dimension version of discrete convolution but the subtraction is reversed. We also divide by the amount of "energy" in the picture to normalize the output. The larger the match, the higher the correlation. We get a perfect match at $d(H_1, H_2) = 1$.

- **Intersection**
$$d(H_1, H_2) = \sum_k min(H_1(k), H_2(k))$$
The larger the metric the better the match.

There are many famous applications of the SIFT Detector in things like panorama stitching, feature-extraction, and image-analysis. One of it's biggest drawbacks is in changes in 3d view-points. If you take a photo from different angles, the SIFT detector will have trouble detecting the object.

4.13 Overview of Edge Detection

This is the most dense of the sections so far with proper applications, and so, I want to be careful to make sure I quickly recap on the things that are pretty important to remember from this section going forward:

- **Gradient Edge Detector** - you do not need to know anything specific, just note the basic ideas of Gradient-based Edge Detectors, and remember some examples of famous versions such as the Roberts, Prewitt, and Sobel filters.

- **Canny Edge Detector** - you should have a general idea of what is going on in the Canny Edge Detector since many implementations in coding modules require you to manually "program" certain components. The steps are:

  1. Run a Gradient-based filter
  2. Compute magnitudes and angles of gradient
  3. For each resulting pixel, compute number of zero-crossings in a single-dimensional Laplacian (2nd derivative) in the direction of the gradient
  4. Threshold the final result

- **Basic Boundary Detection** - recall the notions of boundary-detection and how it differs from edge-detection (computing objects/segments instead of just what could "possibly" be an edge), and also remember the corner-detection algorithms we used.
- **Hough Transform** - recall the Hough Transform and the Generalized Hough Transform and how a transformation from the image space to the parameter space (in both vector format or in numerical format) can make it easy to find the boundary of an image given we know the input it might look like.
- **SIFT Detector** - recall the ideas of using interest points to detect blobs, the Difference of Gaussians, the Histogram of Oriented Gradients, and a general process of how they all come together so we can broadly detect features. Also recall how we represent the features as a histogram of angles of the gradient.

Most of these should be more than enough to understand edge-detection principles. I won't include a further information section since almost all of this information was taken directly from **here**.

5 Image Stitching

Suppose we have a bunch of photos of a big scene taken in angles from eachother and we want to combine the photos to make one big photo. Think of an angled panorama on your smartphone. This is the point of image-stitching!

5.1 Intro to Image Stitching

As aforementioned, image stitching is the process of stitching various images together in order to make a bigger photo. This is especially important in the case where the images are angled, and more importantly, contain some type of repeat/overlap within them. These algorithms are important for panoramas, and taking many photos in different angles and trying to present them in a 3d format – take, for example, the Google Maps Street View. Let's divvy the task of image stitching into two separate tasks:

1. **Overlying** - this is the process of taking many images and overlaying them to produce a single image. We often do this by taking SIFT descriptors (the histogram of angles of gradients of a particular region as described in the earlier section), and find where similar SIFT descriptors lie in the other photo(s). We then need to find some type of transformation to warp the initial image to neatly match with the other images.
2. **Blending** - normally the above process unfortunately results in an image with lines going through it at the points of "blending" which often makes the image look less pleasant and divided. Thus, we want to blend out those lines in order for the image to look more "cohesive".

5.2 Affine and Projective Transformations

Please ensure you are familiar with typical linear algebra transformations using 2x2 matrices such as:

- Rotations
- Dilations
- Skews

- *Make sure you also understand how a linear transformation is equivalent to a change of basis.*

One major drawback of 2x2 matrices is that translation is impossible. There is simply no "space" to add that constant in the matrix. Let's introduce the idea of **homoegeneous coordinates** in order to do this. Define the homogeneous representation of 2D point $p = (x, y)$ is a 3D point $\bar{p} = (\bar{x}, \bar{y}, \bar{z})$ where the third coordinate $\bar{z} \neq 0$ is fictitious such that:

$$x = \frac{\bar{x}}{z}$$

$$y = \frac{\bar{y}}{z}$$

Imagine a 3x3 matrix denoted as A operating on this homogeneous vector:

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

$$Z = AY = \begin{pmatrix} a\bar{x} + b\bar{y} + c\bar{z} \\ d\bar{x} + e\bar{y} + f\bar{z} \\ g\bar{x} + h\bar{y} + j\bar{z} \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix} \begin{pmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{pmatrix}$$

Now, let's try and construct a nice analog to the vector (x, y) – obviously in order to make $\bar{x}$ and $\bar{y}$ the same, we need $z = 1$, and so, the homogeneous coordinate equivalent to (x, y) would be $(x, y, 1)$. This means the transformation A is defined:

$$Z = AY = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix} \begin{pmatrix} \bar{x} \\ \bar{y} \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + c \\ dx + ey + f \\ gx + hy + j \end{pmatrix}$$

Notice that now we are able to add translation into our transformations. Importantly, many times translations shouldn't change the "multiplicity" of the coordinate by affecting the z factor, so when making a translation or rotation or anything of the sort, we automatically define the last rows to be $(0, 0, 1)$:

$$Z = AY = \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \bar{x} \\ \bar{y} \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{pmatrix}$$

We define these types of transformations as **affine transformations** – this means that the transformation preserves lines and parallelism while not necessarily preserving angles or distances/lengths. If we let the bottom row of the transformation NOT be $(0, 0, 1)$, then we have a **projection transformation** or projective transformation. This means that we are projecting the vectors into a different subspace. For sake of being thorough, I will write the basic forms of many "typical" affine transformations in the homoegenous coordinate method so we can see how it works:

- **Scaling**: scaling a given vector by certain constant(s)

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Skew**: smearing the image in a given direction (for this transformation, in the y direction, take the transposition to do the opposite effect)

$$\begin{pmatrix} 1 & m_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Translation**: moving the "origin" by some given constant (or moving all the set of points by some given constant) – in this case constant is referring to some vector (t_x, t_y).

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

- **Rotation**: assume the agnle we are rotating is θ, then the transformation is very similar to the 2x2 version:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Note that we can compose the transformations by multiplying to achieve the combined effect in **reverse order**. If you want to skew then scale then translate then rotate, you just multiply rotate to translate to scale to skew and then multiply that all to the vector.

5.3 Computing Homography Matrices

Let's first define **homography** as the idea of relating any two images of a particular plane taken from different angles. The matrix defined to "relate" the two is called the **homography matrix**! This enables us to perform image stitching of two images taken in different "views" of the same exact plane.

I mentioned the pinhole projection model and lens formulation model earlier, so let's take advantage of those to "convert" from one image space to another in order to combine the two images! We need to first create some assumptions:

- The scene is far away from us. Earth is obviously not a plane, and so is no scene, but if we imagine the scene as being "far enough away from us", we can claim it's a "plane at infinity".
- The SIFT detectors are relatively accurate and are easy to differentiate. If we have very blurry images or images that simply don't have enough SIFT information, this type of image stitching technique simply will not work.

The goal of the homography is to construct a matrix to take the homogeneous coordinates of one image and "warp" them into the coordinate space of another so let's write out this equation with a matrix h:

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} h_11 & h_12 & h_13 \\ h_21 & h_22 & h_23 \\ h_31 & h_32 & h_33 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

Note that subscript s denotes source image coordinates and subscript d denotes destination image coordinates. Also recall that any scalar multiple of the homogeneous coordinates are all **equal** which implies that there is a restriction on the matrix in some form because it won't be injective since it's possible for the results to all be the same. We deal with this by **normalizing the matrix**, and this is done by allowing a choice of 8 different parameters for the matrix and a final parameter to be picked to normalize the magnitude of all the matrix entries to 1. **This means that we have 9 parameters in the matrix but 8 degrees of freedom**.

It also turns out we only need 4 matching SIFT points in the images in order to compute the matrix because we have 8 degrees of freedom (by Fundamental Theorem of Algebra, we need at least 8 equations as a result, and so, the 4 matching points in both images count as 8 total). *I will begin doing quite a bit of math to explain the computation of the matrix, feel free to skip and get to the final result as well if that's what's more important to you.*

$$\begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = \begin{bmatrix} h_1 1 & h_1 2 & h_1 3 \\ h_2 1 & h_2 2 & h_2 3 \\ h_3 1 & h_3 2 & h_3 3 \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

Let's use the notions of homogeneous coordinates to compute the coordinates x and y for a particular instance i of corresponding points

$$x_d^{(i)} = \frac{\tilde{x}_d^i}{\tilde{z}_d^i} = \frac{h_{11}x_s^i + h_{12}y_s^i + h_{13}}{h_{31}x_s^i + h_{32}y_s^i + h_{33}}$$

$$y_d^{(i)} = \frac{\tilde{y}_d^i}{\tilde{z}_d^i} = \frac{h_{21}x_s^i + h_{22}y_s^i + h_{23}}{h_{31}x_s^i + h_{32}y_s^i + h_{33}}$$

We can rearrange this into a matrix equation by subtracting the left side from the right side and obtain:

$$\begin{bmatrix} x_s^i & y_s^i & 1 & 0 & 0 & 0 & -x_d^ix_s^i & -x_d^iy_s^i & -x_d^i \\ 0 & 0 & 0 & x_s^iy_s^i & 1 & -y_d^ix_s^i & -y_d^iy_s^i & -y_d^i & \end{bmatrix} \begin{bmatrix} h_1 1 \\ h_1 2 \\ h_1 3 \\ h_2 1 \\ h_2 2 \\ h_2 3 \\ h_3 1 \\ h_3 2 \\ h_3 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that this equation is for a **single** SIFT interest matching point which is why it's actually impossible to solve (there are too many unknowns). Note how when it's visualized like this, it might make sense why I said 4 interest points are enough to compute this matrix. For each interest point, we can just take the above equation and stack it on top of eachother (take the rows and just keep stacking). We can then get, in matrix form, the below equation:

$$Ah = 0 \, \text{such that} \, |h|^2 = 1$$

Remember that the h being normalized is important to ensure that we don't get repeat values from the homogeneous coordinates. Also note that this type of problem is *very* famous across math/science and can be modeled as the **Constrained Least Squares**:

$$\text{min of } |Ah|^2 \text{ such that } |h|^2 = 1$$

$$|Ah|^2 = (Ah)^T(Ah) = h^TA^TAh$$

$$|h|^2 = h^Th = 1$$

Let's define loss function $L(h, \lambda)$ such that:

$$L(h, \lambda) = h^TA^TAh - \lambda(h^Th - 1)$$

We can take the derivative of the loss function with respect to h and set it equal to 0 to find the minimum and obtain:

$$2A^T Ah - 2\lambda h = 0$$

$$A^T Ah = \lambda h$$

Note that this is basically asking us to compute the eigenvalues of $A^T A$ and the h we are looking for is the eigenvector corresponding to the smallest λ of the matrix $A^T A$.

Quick recap of the section because there is so much math: We care about taking an image and warping it to another images coordinate space using the SIFT Descriptor features. In order to do this, we need to assemble a particular matrix A which takes in each of the features locations in the source image and destination image, and compute the eigenvector corresponding to the smallest eigenvalue of $A^T A$. We can take that eigenvector and then use it to reconstruct the matrix that warps from the two image spaces by taking the first 3 entries and making them row 1, second 3 entries making them row 2, and so on so forth to finish the matrix.

5.4 Outliers with RANSAC

One of the biggest issues we'll have with SIFT Descriptors that I mentioned earlier is that there will be outliers. Often times we will detect features in one image that aren't prevalent in the other, or features that don't really exist. Recall that the SIFT Detector uses the notion of "blobs" to find where general interest points are, but sometimes in different lightings and different positions, certain things just appear like "blobs" when they aren't, and certain things won't. Let's discuss the **Random Sample Consensus** approach to dealing with outliers – it's also called **RANSAC**:

1. Randomly choose s samples. Typically s is the minimum samples to fit a model (in this case, we need 4)
2. Fit the model to the randomly chosen samples.
3. Count the number M of data points that **fit** the model within some type of measure of error e – recall we can use the constrained least squares approach discussed earlier to find the closest we can get to the exact warping. We can pick e depending on the pixelation and what we want for the model.
4. We repeat the above steps N times where N represents the number of interest points.
5. We choose the model that has the largest number M of datapoints that fit the model.

This basically just randomly selects the minimum of points necessary a certain number of times to pick which random selection of minimum points works best for our use case. This is a really good approach to use for SIFT features.

5.5 Warping and Blending Images

Warping an image is slightly harder than you would think – often times, the coordinates that we get as a result of the trasnformation don't really fit in an image frame because they might be non-integers/decimals. Instead, we use an idea termed **backwards warping** which is where we create an image-space for the resulting image (after warping), and take the inverse of the matrix we found prior.

Quick mathematical note: although it might appear that $Ah = 0$ might make it impossible for h to have an inverse, recall that h the vector and H the matrix are two separate entities. The *matrix is composed of separating the vector into sections of 3. The matrix H is required to be an invertible matrix because the transformation is a projection onto a separate plane (not line or point), which means there can be no reduction in dimension or anything excess of the trivial solution in the null space.*

Thus, we take the inverse of the transformation we want to apply, and compute a weighted average of the pixels at that index. So, if we take pixel p and perform the inverse and get something that lies between pixels a,b, and c, then we weighted-average those up (depending on how close the inverse of p is to those particular values) and obtain what the initial pixel was. This technique is essentially combining the **Nearest Neighbors** and **Interpolation** techniques while intelligently incorporating the inverse of the transformation!

I mentioned earlier, but there are often hard seams due to exposure differences and **vignetting** which is when the outer parts of an image appear duller compared to the center because of the light that enters through the lens. This can be solved by using a variant on a distance transform which basically weights images closer to the edges of the image lower. What this means is that when overlaying the images, the pixels that contribute the most are the ones *distant* from the edges which allows for effects such as exposure-differences and vignetting to become mostly negligible. Doing this all allows us to combine images neatly and seamlessly.

6 Facial Detection and Recognition

This section is about the computer vision behind facial recognition, and will briefly dive into some other topics pertaining to facial recognition. Note that this will predominantly **NOT** be including deep learning techniques.

6.1 Intro to Facial Detection and Recognition

Recall that facial detection and recognition although used interchangeably in non-technical settings, are very different:

- **Facial Detection** - facial detection is the idea of finding where a face is located in an image. This is very similar to what you might see in a camera which highlights "boxes" on the face.
- **Facial Recognition** - facial recognition is identifying a face and either labelling it with a database with different photos or computing whether it matches one particular photo.

Here are some famous applications for **facial detection** (not facial recognition):

- **Mobile Cameras** - since mobile cameras do not have the same parameters that a DSLR camera might have (ISO, Shutter Speed, and f-Stop), mobile cameras will often detect where faces are in order to adjust the parameters *automatically* so that faces are in focus and appear their absolute best.
- **Facial Recognition** - particularly important for surveillance and biometrics. This will be discussed later with many practical applications of facial recognition algorithms.
- **Searching + Miscellaneous Use** - it's very common you want to either search through different images to find "faces" or things that match with "faces" (think for example searching on google), or analyzing certain attributes of peoples faces (such as age, gender, race, etc...) for certain tasks.

6.2 Haar Features for Facial Detection

I have a section on the Discrete Haar Transform which talks about using Haar Filters for compression by creating corresponding low-pass and high-pass filters. I'll re-explain a part of this by revisiting how Haar Features are constructed:

1. **Haar Wavelets** - these are wavelet functions which look sort of like square functions - 0 everywhere but a domain of x to y - for half of the domain of x to y the wavelet is positive 1 and the other half, it is -1. It is worth noting that sometimes these values are actually divided by $\sqrt{2}$ in order to normalize the total integral of the function to be 1.
2. **Haar Filter** - from these wavelets, we can develop a filter which *basically* is the same as applying a convolution with various different Haar Wavelets at different "depths" or "levels". Read more in the Discrete Haar Transform section to get a better idea.
3. **Haar Features** - now, we basically take the 1 dimensional Haar constructions from above which are wavelets, and apply them in two dimensions. We take the resulting different 2-dimensional filters and convolve them with the surrounding region of *every* pixel. We normally tend to pick certain versions of these that help us distinguish for faces (or for different objects in other cases), and call the features that we select, the Haar Features. It looks like rectangular boxes of black and white for the most part.

Some important things to know about Haar features are that they are a *single* vector containing several different numbers. Recall that the Discrete Haar Transform returns vectors of vectors (tensors) because it is computing the filters over and over – in this case, we aren't doing this so we get a vector composed of scalars which is the result of performing convolution with the Haar 2-dimensional filters. At certain points, these filters actually closely resemble many of the derivative-based filters we used prior such as the **gradient x-direction** filter, the **gradient y-direction** filter, and the **Laplacian/2nd derivative filter**.

You might be wondering how these Haar filters are particularly helpful. The reason is because it is identifying some of the symmetries only prevalent in faces and not many other objects:

- Many faces are symmetric across the horizontal axis, so a Haar Filter which exploits that could be used for face detection.
- Many faces, however, are NOT symmetric across the horizontal axis. A Haar Filter which exploits that could *also* be used for face detection because it can tell us that something *isn't* a face.
- Note that things get a LOT more complicated with also identifying other features (for example eye-color being darker than skin-color, or eyebrows being darker than teeth, etc).

In general, the Haar features let us exploit certain symmetries that really are only prevalent in faces and not many other objects. This is also partially why the Haar-feature system won't work for faces that are tilted, or for faces that are occluded.

6.3 Integral Images

A huge concern with facial recognition is efficiency. We want the algo to be efficient because face detection, particularly for many of the applications I mentioned, requires *VERY* fast processing time. Since Haar features are like "blankets" (similar to convolution kernels) where we try and sum the pixels of certain areas and subtract the sum of pixels in different areas (the way the Haar wavelet function convolved sums something over a given region and subtracts something over a given region), we want to have a construction which takes advantage of this.

Here is where we use the **integral image** which is essentially a reconstruction of the image where each "value" is represented by the sum of pixels above and to the left of the pixel. You can read more about this in Shree K Nayar's video series since it goes more into depth about how this is calculated, but using the integral image makes it so that computing the sums over large swaths of the image reduces to a constant time problem instead (constant number of sums since we already have the sums in given locations). This makes it significantly easier to compute the Haar features and construct the vector significantly faster.

NOTE: taking an image and turning it into this "sum of previous parts" is actually an $O(n)$ operation which is relatively straightforward to do in a raster-method. Watch the video to learn more about its practical implementation.

6.4 Nearest Neighbor Classifier and Support Vector Machines

The Nearest-Neighbor-Classifier is a paradigm for classifying feature vectors. Given some type of vector space where we plot the feature vectors, the **nearest neighbor classifier** will find the closest vector in terms of Euclidean distance to the initial vector. This means that we take the vector, find the nearest vector that is already labelled/classified, and we assign the vector to that vector's classification.

You might note that this is very redundant – why do we have to compute the nearest vector? We can introduce the notion of **decision boundaries** which are, usually, surfaces in the vector space (note that they can and usually are of very high dimensions). If a feature is on one side of the surface, then it is classified as a face, while if the feature is on the other, it is classified as *NOT* a face. We construct such (linear) surfaces using **Support Vector Machines.** A key assumption to note for this, is that the surface for faces vs non-faces is almost always going to be *linear*, and so we can use support vector machines freely.

I will omit much of the math here since I think it's not particularly helpful. The goal is to optimize a plane to separate two separate "sets" of points. We do this by trying to find all possible hyperplanes (higher dimensional planes) that meet the criteria of separating the points, and then computing the largest **margin**. The margin is the largest n-th dimensional rectangular prism you can make by slowly extending the separating hyperplane outward. We want to pick the hyperplane with the largest margin, so that way it is the most "accurate" (basically, it is the surface that evenly splits the barriers of the two sets of points).

If interested, search up a visual depiction of Support Vector Machines and their associated Margins in order to further understand how they are defined.

6.5 Viola-Jones Facial Detection Algorithm

The Viola-Jones algorithm is by far the most famous and widely used algorithm for detecting faces in some type of image. This is especially shocking given that it doesn't use *any* deep learning techniques. It uses some of the things mentioned above, but let's thoroughly outline the algorithm here:

1. **Integral Image Calculator** - this is the exact same as what was mentioned earlier – we want to calculate the integral image in order to make the Haar-features easier to compute.
2. **Haar-like Feature Extraction** - we want to extract Haar features as mentioned earlier because they are extremely efficient at finding features/attributes of faces.

3. **Feature-Selection** - this is the process of selecting features from the image – this is typically done with a machine learning approach.
4. **AdaBoost** - most of the Haar features are considered “weak classifiers” because they aren’t (on their own) good enough to classify faces versus non-faces. Thus, we also use adaboost in order to take some of the weak classifiers and combine them (through using machine learning again) to create even strong classifiers.
5. **Cascade of Classifiers** - We order the features based on how “strong” they all are combined that way we can apply them continuously and allow for **early rejection** – that is, denoting the non-faces as legitimately NOT faces very early on.
6. **Thresholding** - in most machine learning models, the output is a probability that something will happen thus we threshold the probability to decide if it’s “likely enough” that the object at hand is a face.

Obviously the algorithm is important to understand, but the main **very** important thing which the Viola-Jones algorithm introduced to the world is the idea of **boosting** through the AdaBoost system. It is widely used in computer vision and machine learning as a technique (which I won’t explain) that can combine many weak classifiers into a strong single one.

6.6 Eigenfaces Algorithm and Deep Learning for Facial Recognition

This is, a very brief overview of the above techniques for facial recognition. Note that I don’t go in depth because these techniques are mostly related to deep learning and things of that nature – it is less of a computer vision task and more of a deep learning task to check if two faces are the same and to analyze what features make faces the same. Note that pretty much both methods first require that facial **detection** algorithms are run first to figure out where the faces are located.

- **Eigenfaces** - the broad logic of this method is to use singular value decomposition, eigenvalue/spectral decomposition, and principal components analysis in order to compute the main components of an image vector. We typically represent an image (the part of an image with a face in it) as a high dimensional vector (tensor rather) of dimension N and M – NxM vector. By doing this, we find the components that capture the highest variation in the image, and use those to determine future images.
- **Deep Learning** - many deep learning techniques for facial recognition exist and to learn more about them, you can read the Szeliski textbook on Computer Vision. One famous version of this is **DeepFace** which is a facial recognition software developed by Facebook researchers. It uses **frontalization** in order to make a face “face” forward. The endgoal is to make sure that the image is cropped and facing forward. Then, it uses various deep learning methods in order to analyze the image and determine some characteristics about the face to match which it matches to a **vector embedding** – this is a vector-representation of the image (this same term is used in NLP for a vector representation of a word). Furthermore, we develop a similarity metric which takes the two vectors containing important features about the face, and use the similarity metric to determine how likely it is that the two faces are the same.

Broadly speaking, the issue of facial recognition is extremely challenging and there exist many deep learning based algorithms to do it. If you understand how Convolutional Neural Networks function, and the ideas of supervised learning, pooling, activation functions, loss functions, and backpropagation (sort of the basic building blocks of multilayer perceptron neural networks and deep learning), then read about DeepFace online to understand more about how it is used and the detailed workings of how it works.

7 Deep Learning Techniques

This section is going to a huge deep dive in deep learning techniques. Many of other techniques such as Nearest-Neighbor Classifiers and things like that have been mentioned earlier, so I will simply start out with things I'm assuming you guys to know. The vast majority of this chapter will be focussed on things like Convolutional Neural Networks and other computer-vision focussed deep learning tasks. A lot of this will be based on "Computer Vision using Deep Learning" by Vaibhav Verdan.

7.1 Convolution, Pooling, and Activation

This section is going to talk about convolution, pooling, and activation functions that are commonly used in CNNs.

- **Convolution** - convolution, as mentioned before, is the idea of taking a kernel and sliding it over some type of region to get some information – it is often used in many gradient-based filters, and is used as a way to extract different features which can then be plugged inside the model.
- **Pooling** - pooling, also mentioned earlier, is the idea of agglomerating certain regions of an image into one single value to make it easier to process in a neural network.
- **Activation** - these typically involve ReLU (Rectified Linear Units) which will basically make all negative values 0 and retains all positive values to be the same.

Note that all of these together tend to form the typical Convolutional Neural Network (CNN) which we use today for a lot of image prediction techniques and things of the sort. However, we can also progress further and discuss other specific network architectures.

7.2 LeNet Architecture

This is a specific type of architecture of a convolutional neural network used explicitly in image processing and computer vision developed back in the 1980s timeframe. It was tailored for the MNIST database (so understanding and detecting handwritten digits) – more importantly, it was also designed to be easily used on a CPU (instead of needing some dedicated other GPU hardware for it to work). This was designed back when max-pooling wasn't a concept so average-pooling was used instead. I will first explain the LeNET type 4 architecture and explain what the steps are and briefly explain about some differences between this architecture and some others and why one would pick one over the other:

1. **Input Layer** - normally a 32 by 32 input image
2. **Convolutional Layer** - four 24x24 convolution layer
3. **Pooling Layer** - normally an average pooling layer (could also be max if you wanted that instead)
4. **Convolutional Layer** - 16 12x12 convolution layer
5. **Pooling Layer** - average pooling layers run twice

6. **Output Layer** - the display tensor reflects certain characteristics based on the image input

The above loosely describes the **LeNet 4 Architecture**. But, we also have the **LeNet 5 Architecture** as well which adds more convolutional layers, more types of pooling layers, and differing types of activation functions. Overall, it is the more widely used between the two and is the one made more recently. We also have the idea of a **Boosted LeNet Architecture** which is typically applied to LeNet 4, but works with pretty much anything. This is very similar to the **Adaboost** technology found in the Viola-Jones facial detection algorithm – it will find and compute the weak classifiers and combine them to make them a stronger predictor of whatever the model is trying to predict.

7.3 AlexNet and VGG Neural Networks

These are some other models popularly used in the field of computer vision with more complicated images and designs. The architecture uses a series of convolutions, max-poolings (instead of average-poolings found in the previous section on LeNet architecture), and various fully connected layers equipped with the RELU activation function to add nonlinearity. Dropout layers and data augmentation was used in order to fight overfitting in the network.

Another extremely famous set of neural networks used for computer vision is the VGG – VGG16 and VGG19 are the two main types:

- VGG16 - this is the simpler version of the VGG neural network and includes 3x3 convolution layers and 2x2 pooling layers throughout the network. It's often used as a benchmark for many image-classification algorithms for particular use-cases to see if it is "better" than the average deep-learning model.
  - Contains 16 layers
  - Size in terms of FC layers is 533mb
  - Lighter model
  - Preferred for smaller datasets
- VGG19 - this is the same thing as above but significantly more stricter it is due to the fact that in the general business world
  - Contains 19 layers
  - Size in terms of FC layers is 574MB
  - Larger and deeper network
  - Can be used for richer categories as high as 1000 classes

Broadly speaking, the order in which these were introduced (AlexNet, VGG16, VGG19), are the order in which they were created, and the order in least deep and accurate to most deep and accurate. The tradeoff is computational power and memory. That is why the industrial standard has pretty much stuck with VGG16!

7.4 Deep Learning for Object Detection - RCNN & YOLO

To introduce this idea, we can first introduce the notions of moving and bounded boxes – both of these are done by taking boxes and trying to do different types of manipulations to find the object at hand.

Let's define some of the main deep learning approaches used for this task:

- **Region-based CNN (RCNN)** - do not confuse this with Recurrent Convolutional Neural Networks – this is based on the idea of taking different regions of an image, and computing the CNN features of them (by using convolution). We then classify the regions by passing the features into a support vector machine to clarify the presence of objects (specific objects) in the region proposed.
- **Fast RCNN** - this is the same approach as above, but computationally optimized to make it faster. The architecture is very similar but takes the different regions and applies pooling in order to make sure that there are less regions to process (and smaller regions to process).

There are some other types of Region Based Neural Networks as well including the **Faster RCNN** as well, but I won't mention those for sake of simplicity. One of the most famous object detection algorithms also used today is **YOLO - You Only Look Once** - the process works the following way:

1. YOLO divides the input image into an SxS grid - each grid is responsible for predicting only one object so if the center of an object falls in a grid cell, that grid cell is "responsible" for detecting the object.
2. For each of the grid cells, it predicts boundary boxes based on various attributes, and it gives a score (to reflect how likely it is that an object is inside the box).
3. The confidence/score is defined as Probability(object) times IoU (Intersection of Unions).
4. Each grid cell predicts C conditional class probabilities - Pr(Classi — Object). These probabilities are conditioned on the grid cell containing an object. we only predict one set of class probabilities per grid cell, regardless of the number of boxes B.
5. We multiply conditional class probabilities and individual class predictions to be even more certain that an object is located in the box.

Basically, YOLO is trying to create multiple "bounding boxes" of each object within a cell – to calculate the loss, YOLO optimizes for sum-squared error in the output in the model as sum-squared error is generally easy to optimize.

Broadly speaking, YOLO looks at the image only once but in a clever manner, it is just trying to divide the image into boxes and use different loss functions and attributes to determine if the object is located and where the boundary of the object is located. Importantly, here are some things to note about YOLO which is what makes it a very common industrial standard:

- YOLO is **very** fast
- YOLO is **very** simple to use as well
- YOLO also reasons globally about the image when making predictions instead of other region-based techniques because YOLO combines everything together at the end.
- YOLO, because it uses a combination of the different boxes, learns a lot of generalizable information about the representation of certain objects instead of region-based convolutional neural networks which don't necessarily do the same thing.

Another really new but important object detection algorithm is the **Single Shot MultiBox Detector (SSD)** which overcomes slowness in networks to work in real-time object detection by taking the VGG16 architecture but modifying it in order for a single shot to be required to detect multiple images in an object.